

# Supplementary Information for Precision meets speed through an FPGA-based natively-sparse Ising machine for combinatorial optimization

Baijian Yao<sup>1</sup>, Daniel Ebler<sup>\*2</sup>, Xu Shi<sup>1</sup>, Juntao Wang<sup>2</sup>, Fan Zhang<sup>2</sup>, Wei Zhang<sup>†1</sup>, and Jie Sun<sup>‡2</sup>

<sup>1</sup>Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong SAR, China

<sup>2</sup>Leibniz Theory Lab, 2012 Labs, Huawei Technologies Co. Ltd., Hong Kong SAR, China.

## Supplementary Note 1. Simulated bifurcation

A heuristic algorithm that provides high quality approximate solutions to ground-state solutions of the Ising problem is Simulated Bifurcation (SB)<sup>1,2</sup>. In SB, a network of  $N$  oscillators encodes an optimization problem with matrix  $J$  into the interaction of non-linear systems and solves the problem by the temporal evolution of the systems. The oscillators are described by the Hamiltonian

$$H_c(\mathbf{x}, \mathbf{y}, t) = \sum_{i=1}^N \left[ \frac{K}{4} (x_i^2 + y_i^2)^2 - \frac{\mathbf{p}(t)}{2} (x_i^2 - y_i^2) + \frac{\Delta_i}{2} (x_i^2 + y_i^2) \right] - \frac{\xi_0}{2} \sum_{i=1}^N \sum_{j=1}^N J_{i,j} (x_i x_j + y_i y_j), \quad (1)$$

which constitute the dynamical equations

$$\dot{x}_i = \frac{\partial H_c}{\partial y_i} = [K (x_i^2 + y_i^2) + \mathbf{p}(t) + \Delta_i] y_i - \xi_0 \sum_{j=1}^N J_{i,j} y_j \quad (2)$$

$$\dot{y}_i = -\frac{\partial H_c}{\partial x_i} = -[K (x_i^2 + y_i^2) - \mathbf{p}(t) + \Delta_i] x_i + \xi_0 \sum_{j=1}^N J_{i,j} x_j. \quad (3)$$

Here,  $x_i$  and  $y_i$  are a pair of canonical conjugate variables, representing the position and momentum of the oscillator  $i$ ,  $\Delta_i$  and  $\xi_0$  are hyperparameters which can be adjusted by the experimenter.

**Ballistic SB and quantization.** A variant of SB, ballistic SB (bSB), introduces additional non-linearities such as the truncation of spin values<sup>3</sup>. The dynamical equations of bSB with first-order approximation give rise to the following update rule

$$\begin{aligned} \mathbf{y}(t + \Delta t) &= \mathbf{y}(t) + \Delta t \cdot [(-a\mathbf{1} + \mathbf{p}(t)) \odot \mathbf{x} + \xi J \cdot \mathbf{x}(t)], \\ \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t \cdot \mathbf{y}(t + \Delta t). \end{aligned} \quad (4)$$

---

\*Corresponding author: Mail address: ebler.dan@gmail.com

†Corresponding author: Mail address: wei.zhang@ust.hk

‡Corresponding author: Mail address: riosun@gmail.com

We can separate bSB into two parts, the sparse matrix-vector multiplication (SpMV)

$$\mathbf{MV} = \mathbf{J} \cdot \mathbf{x} \quad (5)$$

and time evolution (TE)

$$\begin{aligned} \mathbf{y}(t + \Delta t) &= \mathbf{y}(t) + \Delta t \cdot [(-a\mathbf{1} + \mathbf{p}(t)) \odot \mathbf{x} + \xi \cdot \mathbf{MV}], \\ \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t \cdot \mathbf{y}(t + \Delta t). \end{aligned} \quad (6)$$

The SpMV  $\mathbf{J} \cdot \mathbf{x}$  is the most time-consuming part of the iterations, scaling as  $O(N^2)$  in the number  $N$  of spins.

The algorithmic implementation of bSB is shown in Supplementary Algorithm 1

```

input: Coefficients Matrix of size  $w \times w$ :  $\mathbf{J}$ ,
        Initial position vector: init_x,
        Initial momentum vector: init_y,
        Number of iterations: iter_nums,
        Time evolution parameter:  $dt$ 
output: A final vector of position  $\mathbf{x}$ 

x_comp  $\leftarrow$  init_x
y_comp  $\leftarrow$  init_y
p[t]  $\leftarrow$  linspace(0, 1, iter_nums)
 $\xi \leftarrow (0.5/\sqrt{N})$ 

for  $i \leftarrow 0$  to iter_nums do
    y_comp  $\leftarrow$  y_comp +  $[(-a_1 + \mathbf{p}[t]) \cdot \mathbf{x\_comp} + \xi \cdot (\mathbf{J} \cdot \mathbf{x\_comp})] \cdot dt$ 
    x_comp  $\leftarrow$  x_comp + y_comp  $\cdot dt$ 
    y_comp[x_comp > 1]  $\leftarrow$  0
    x_comp.clamp_(-1, 1)
    sol  $\leftarrow$  x_comp.sign()
     $e \leftarrow -\frac{1}{2} \cdot \mathbf{sol}^T \cdot \mathbf{J} \cdot \mathbf{sol}$ 
    energies.append( $e$ )

```

Supplementary Algorithm 1: ballistic simulated bifurcation

In the main text, qSB was introduced as a quantized version (INT8 precision) of bSB, see Supplementary Algorithm 2 below for the concrete implementation. Lower precision allows for reduced memory requirements and faster processing. Despite the precision, the convergence behavior of qSB is similar to bSB and qSB shows a higher solution accuracy than bSB in most cases, see Supplementary Figure 1. The max-cut values obtained by qSB are 0.3% better than bSB on average ( $G1 - G50$  Gset problems).

**Simulated Bifurcation Benchmark.** As a benchmark metric to evaluate the performance of an Ising machine, time-to-target and time-per-step are commonly used. The time-to-target is typically defined as the runtime required to achieve 99% of the best known solution, which typically involves around 1000 simulation steps. However, in work<sup>1</sup>, it is defined as the runtime to achieve the energy a Hamiltonian neural network (HNN) reaches, which only involves around 20-59 simulation steps. In addition, the time-per-step is defined as the runtime for each simulation step. The relation between time-per-step and time-to-target can be generalized as

$$T_{\text{to-target}} = T_{\text{per-step}} * \text{\#Steps} \quad (7)$$

where  $\text{\#Steps}$  are the number of steps of the algorithm (such as  $\text{\#Steps}_{\text{HNN}}$  around 20-59 and  $\text{\#Steps}_{\text{BestKnown}}$  around 800-1000 steps).

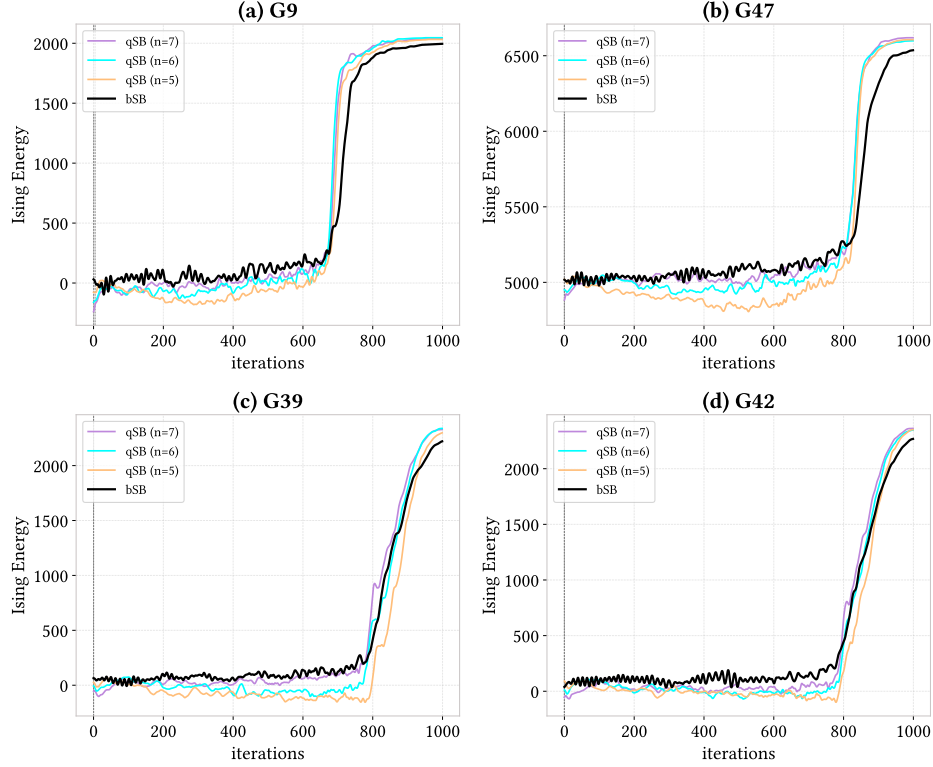
```

input: Coefficients Matrix of size  $w \times w$ :  $J$ ,
        Initial position, momentum vector:  $\mathbf{init\_x}$ ,  $\mathbf{init\_y}$ ,
        Number of iterations: iter_nums,
        Time evolution parameter:  $dt$ 
output: A final vector of position  $\mathbf{x}$ 
scaling factor:  $S = 127$   $\mathbf{x\_comp} \leftarrow \mathbf{init\_x} * S$ 
 $\mathbf{y\_comp} \leftarrow \mathbf{init\_y} * S$ 
 $\mathbf{ps} \leftarrow (-1 + \text{linspace}(0, 1, \text{iter\_nums})) * S$ 
 $\xi \leftarrow (0.5/\sqrt{N})$ 
 $dt \leftarrow 1/4$ 

for  $i \leftarrow 0$  to iter_nums do
     $\mathbf{y\_comp} \leftarrow \mathbf{y\_comp} + [\mathbf{ps}[i] \cdot \mathbf{x\_comp} + \xi \cdot (J \cdot \mathbf{x\_comp})] \cdot dt$ 
     $\mathbf{x\_comp} \leftarrow \mathbf{x\_comp} + \mathbf{y\_comp} \cdot dt$ 
     $\mathbf{y\_comp}[\|\mathbf{x\_comp}\| > S] \leftarrow 0$ 
     $\mathbf{x\_comp} \leftarrow \text{clamp}(\mathbf{x\_comp}, -S, S)$ 
     $\mathbf{sol} \leftarrow \text{sign}(\mathbf{x\_comp})$ 
     $e \leftarrow -\frac{1}{2} \cdot \mathbf{sol}^T \cdot J \cdot \mathbf{sol}$ 
    energies.append( $e$ )

```

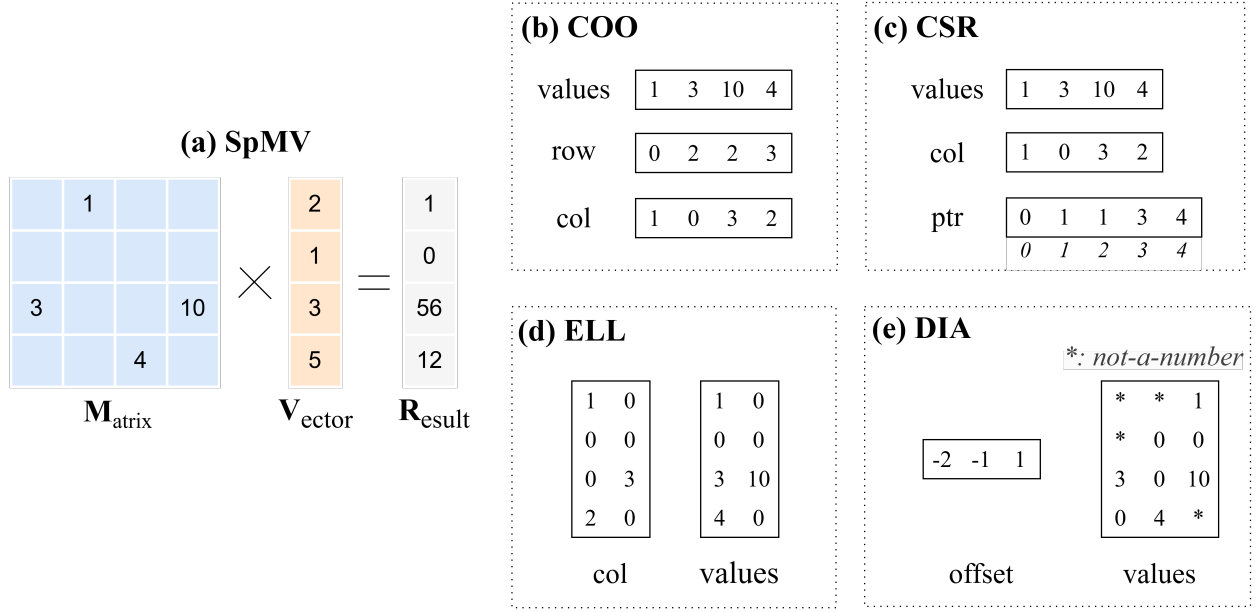
Supplementary Algorithm 2: quantized simulated bifurcation



Supplementary Figure 1: Time-energy graph of qSB for max-cut compared with bSB: (a) G9; (b) G47; (c) G39; (d) G42; (Binary uniform quantization width  $n = 7, 6, 5$ ; step-to-target (STT): the step needed for Ising simulation to reach target value)

## Supplementary Note 2. Sparse Matrix-Vector Multiplication

A sparse matrix is characterized by the majority of its elements being zeros, as illustrated in Supplementary Figure 2(a). The sparsity of a matrix is given by  $S = \frac{N^2 - \text{NNZ}}{N^2}$ , or the density  $D = \frac{\text{NNZ}}{N^2}$ , where  $N$  is the matrix size and NNZ is the number of non-zero elements. The density of a sparse matrix is typically less than 5%. To enhance both storage and operational efficiency for sparse matrices, we introduce four commonly used compression formats<sup>4-6</sup> (Supplementary Figure 2): Coordinate List format (COO), Compressed Sparse Row format (CSR), ELL-PACK (ELL), and diagonal (DIA), together with their variants<sup>7</sup>.



Supplementary Figure 2: Basic SpMV format: (a) An example of a sparse matrix-vector multiplication; (b) The COO format consists of a values array denoting the matrix entries, and row and column (col) arrays which denote the coordinate position of the entries in the matrix; (c) The CSR format replaces the row array with a pointer array ptr, which denotes the global position of the first NNZ in each row; (d) The ELL format consists of col and values matrices which store for each row the column positions in the col matrix, and the corresponding NNZ elements and the values matrix; (e) The DIA format consists of an offset array, denoting the shift of the diagonals with NNZ relative to the main diagonal, and a value matrix denoting the positions of the NNZ within the diagonals.

**COO.** As shown in Supplementary Figure 2(b), the COO format comprises three arrays: row index, column index, and value, representing coordinates and values of non-zero entries of a matrix.

The way to evaluate the storage occupancy of the COO matrix is as follows. Assuming each row index and column index require  $B_{\text{idx}}$  bits, and each non-zero element requires  $B_{\text{val}}$  bits, the storage occupancy  $S_{\text{COO}}$  can be calculated as

$$S_{\text{COO}} = (2 \cdot B_{\text{idx}} + B_{\text{val}}) \cdot \text{NNZ}.$$

The COO format was shown to be a generally applicable good sparse data storage format. However, there are some limitations given its simplicity. Firstly, if a sparse matrix contains many NNZs sharing the same rows or columns, storing the same row or column index for each NNZ leads to significant redundant storage and memory access overhead. Secondly, COO-based SpMV algorithms typically use a task division strategy with an equal number of NNZ. This requires atomic operations to merge local results, which significantly impacts SpMV performance and creates additional hardware logic.

**CSR.** The CSR format optimizes the COO format by compressing the row indices array while keeping the other two arrays unchanged. A row pointer array *ptr* is used to store the global position of the first NNZ in each row within col and value arrays. Supplementary Figure 2(c) illustrates the CSR *ptr* encoding for the matrix in the left. The storage occupancy  $S_{\text{CSR}}$  of the CSR format is given by

$$S_{\text{CSR}} = (N + 1) \cdot B_{\text{idx}} + \text{NNZ} \cdot (B_{\text{val}} + B_{\text{idx}}).$$

The CSR format addresses the drawback of the COO format for row-centered matrix and is less sensitive to the data distribution compared to the formats introduced later. ELL and DIA formats. The CSR format is widely used in practical applications. However, for irregular sparse matrices with a significant variation in the NNZ per row, it is difficult to develop CSR SpMV scheduling algorithms to resolve load imbalance when performing row-wise task partitioning.

**ELL.** The ELL format consists of two 2-dimensional arrays of the same size. Col stores the column indices and values the corresponding entries of the matrix. In Supplementary Figure 2(d), all NNZs per row of a sparse matrix are shifted to the leftmost positions. In addition, the width of the two arrays in the ELL format is determined by the longest row width ( $W_{\text{max}}$ ); padding zeros is necessary for rows shorter than  $W_{\text{max}}$ . The 2D-array structure retains the row indices of NNZs with the row index of the array. Therefore, the storage occupancy of the ELL format is given by

$$S_{\text{ELL}} = (B_{\text{idx}} + B_{\text{val}}) \times 2 \cdot N \cdot W_{\text{max}}.$$

An obvious flaw of the ELL format is that the storage occupancy is sensitive to  $W_{\text{max}}$ , which could lead to significant numbers of zero padding. In contrast, if the row width is almost the same without zero padding, ELL is efficient.

**DIA.** The DIA format arranges NNZs in a diagonal-wise pattern and consists of two arrays: offset, which stores the offset of each non-zero diagonal, and values, which stores the corresponding non-zero entries. In Supplementary Figure 2(e), the main diagonal typically has an offset of 0, while the diagonals in the lower triangular region have negative offsets and those in the upper triangular region have positive offsets. For a sparse matrix with  $N_{\text{dia}}$  diagonals that have at least one NNZ, the storage occupancy is given by

$$S_{\text{DIA}} = N_{\text{dia}} \cdot N \cdot B_{\text{val}} + N_{\text{dia}} \cdot B_{\text{idx}}.$$

The DIA format is well-suited for matrices where NNZs are primarily concentrated diagonally. Otherwise, the padding of numerous zeros will significantly increase the storage overhead. Given that the number of elements in each diagonal is not fixed, some diagonals have less entries than the size of the values array. Hence, elements \* that stand for not-a-number are included.

**Why COO.** As described above, the performance of the SpMV formats varies depending on the shape of the matrix. In this regard, COO is the most consistent format when tackling matrices without knowledge of the shape. In addition, COO is easy to extend to optimization techniques, such as tilting, slicing, and bit-mapping (see discussion below), to address shortcomings in specific use cases. Furthermore, COO can easily adapt to memory constraints and stream transmission because other formats have arrays of different sizes. Work<sup>8</sup> indicates that COO consistently outperforms other SpMV storage formats on average in various scenarios.

**Variants of SpMV formats.** To address certain shortcomings of the basic formats introduced above, several variants of the formats have been proposed, see Supplementary Table 1. Given the extensive variety of existing formats, here we focus on key optimization techniques rather than detailing each variant individually.

In Supplementary Table 2, regular slicing partitions the matrix along a single dimension (either rows or columns), so that elements with the same row or column can share the same indexed values. This simplifies

the access patterns. Regular blocking (or tiling) decomposes the matrix into two-dimensional sub-blocks, storing non-zero blocks in a base format (e.g., CSR or COO). This approach reduces memory overhead by encoding block-relative positions rather than global indices.

Irregular compression techniques relax the constraints imposed by row-, column-, or block-based structures, making them particularly effective for matrices with highly irregular non-zero distributions. Bit/byte-level compression further optimizes storage by applying delta coding, run-length encoding, or bitmasking to index arrays in conventional sparse formats.

Finally, hybrid encoding dynamically selects compression strategies based on local matrix structures, significantly reducing memory overhead and eliminating zero-padding inefficiencies in formats such as ELL, SELL, and BCSR. The present work focuses solely on the basic COO format and leaves the investigation of variants to future work.

Supplementary Table 1: Latest variants of SpMV formats. Mixed basic formats denote the case where variants use two or more basic formats as reference.

| basic format | variant format                                                  |
|--------------|-----------------------------------------------------------------|
| COO          | BCCO/BCCOO/BCCOO+ <sup>9</sup>                                  |
| CSR          | BRO-CSR <sup>10</sup> , CSR2 <sup>11</sup> , VBSF <sup>12</sup> |
| ELL          | SELL <sup>13</sup>                                              |
| DIA          | HDIA <sup>14</sup>                                              |
| Mixed        | AMB <sup>15</sup> , BCE <sup>16</sup>                           |

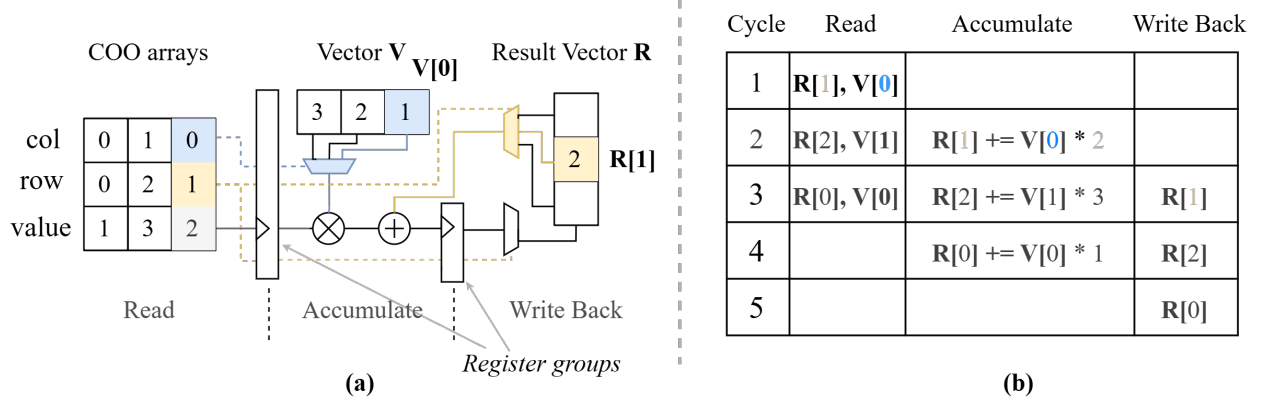
Supplementary Table 2: Commonly used format optimizing techniques. Regular slicing partitions the matrix along column or row direction; Regular blocking decomposes the matrix into sub-blocks; Irregular representing relaxes the constraints imposed by row-, column-, or block-based structures; Bit/byte-level compression applies delta coding, run-length encoding, or bitmasking on the matrix; Hybrid encoding dynamically selects compression strategies based on local matrix structures.

| basic format           | variant format                                                                                              |
|------------------------|-------------------------------------------------------------------------------------------------------------|
| Regular slicing        | Sliced COO <sup>17</sup> , Sliced CSR <sup>18</sup> , Sliced ELL <sup>13</sup> , Sliced DIA <sup>19</sup>   |
| Regular blocking       | Blocked CSR <sup>6</sup> , Blocked COO <sup>9</sup> , Blocked DIA <sup>20</sup> , Blocked ELL <sup>21</sup> |
| Irregular representing | CSR5 <sup>22</sup> , CSR2 <sup>11</sup> , CSRX <sup>23</sup>                                                |
| Bit/Byte compressing   | Bitmask <sup>24</sup>                                                                                       |
| Hybrid combining       | COO+ELL <sup>25</sup> , CSR+ELL <sup>26</sup> , COO+CSR <sup>27</sup>                                       |

**SpMV Basic Computation Pattern.** The common hardware implementation for COO-based SpMV computation flow is shown in Supplementary Figure 3(a). Here, a concrete example of a matrix-vector multiplication is illustrated, with a  $3 \times 3$  matrix stored in COO format and example vector  $\mathbf{V} = (3, 2, 1)$ . The pipeline can be divided into 3 stages: *Read*, *Accumulate*, and *WriteBack*. The COO tuples (*row*, *col*, *value*) are processed by the pipeline one by one, starting with (0, 1, 2). In the *Read* stage, the *row* of a COO array is used to select a value in the result vector  $\mathbf{R}[\text{row}]$  (yellow line,  $\mathbf{R}[1]$  – initially, the value of  $\mathbf{R}[1]$  is zero) and the *col* is used to select the value in the multiplied vector  $\mathbf{V}[\text{col}]$  (blue line,  $\mathbf{V}[0]$ ). In the *Accumulate* stage, the selected  $\mathbf{V}[\text{col}]$  is multiplied by the *value* and used to update  $\mathbf{R}[\text{row}]$  by addition (in the example,  $\mathbf{R}[1] += \mathbf{V}[0] * 2$ ), implemented by the multiplier-adder (x, and + gates) structure in Supplementary Figure 3(a). In the *WriteBack* stage, the new  $\mathbf{R}[\text{row}]$  is written back to the result vector based on the *row* ( $\mathbf{R}[1] = 2$ ). In between each stage, there is a register group to pipeline the hardware. They act as temporary storage, passing the outputs of one stage (e.g., the values and the row/col indices) to the next stage on the next clock cycle. This isolation allows each stage to work on a different tuple simultaneously without

interference, enabling the overlapping execution visualized in Supplementary Figure 3(b). For instance, while the *Accumulate* stage is processing tuple  $N$ , the *Read* stage can be fetching the data for tuple  $N + 1$ .

Supplementary Figure 3(b) gives a timeline showing the pipeline’s operation over multiple clock cycles in each row. In cycle 1, tuple 1 is read. In cycle 2, tuple 2 is read, and tuple 1 is accumulated. In cycle 3, tuple 3 is read, tuple 2 is accumulated, and tuple 1 is written back. Until cycle 6, the whole COO arrays are calculated.

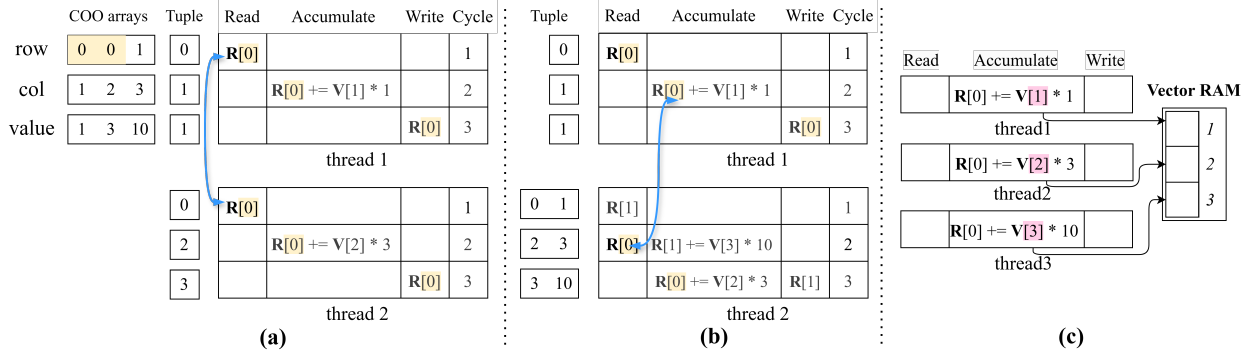


Supplementary Figure 3: Pipelined COO SpMV compute pattern: (a) Pipelined computing flow with stage *Read* (read result vector value  $R[row]$ ,  $V[col]$ ), *Accumulate* (add matrix multiplication result  $V[col] * value$  to result vector  $R[row]$ ), and *WriteBack* (write new result back on  $R[row]$ ). The stages are pipelined, meaning that when stage *Accumulate* is adding a result vector, the *Read* stage can read a new result vector value; (b) Simple compute pipeline implementation; The blue part highlights *col* array related logic (read vector for multiplication); The yellow part highlights *row* array related logic (read result vector for add back); The gray part highlights *value* array related logic (multiplication)

**SpMV Data Dependency.** Upon analyzing the above compute pattern in terms of parallelism, data dependency, and storage resource utilization, several intrinsic SpMV hazards (data conflicts) are identified, as discussed in the following (see Supplementary Figure 4).

1. **Data hazard.** In SpMV, a data hazard occurs when multiple calculation units (PE threads in our design) read, accumulate and write the same vector value. This includes both the intra-stage hazard and the inter-stage read-after-write (RAW) hazard. An intra-stage hazard occurs when different threads read and update the same result vector  $R[row]$  at the same pipeline stage. In Supplementary Figure 4(a), two threads read  $R[0]$  in stage 1 and perform calculations separately, while the two values should be added together. An inter-stage RAW hazard occurs when the updated value has been read out at the next pipeline stage before the calculation result from the previous pipeline stage is written back. In Supplementary Figure 4(b), thread 2 fetches  $R[0]$  while thread 1 accumulates  $R[0]$ , resulting in thread 1 reading an incorrect value.
2. **Structural hazard.** A structural hazard can be defined as a hazard caused by multiple threads attempting to write to the same Result RAM. Considering that when there are multiple threads, each thread can write the SpMV Result RAM. In Supplementary Figure 4(c), three threads access the same vector RAM for vector  $V[col]$ . However, even the most complex Block RAM (BRAM) in Xilinx FPGA has at most two ports for writing at the same time. Therefore, implementing the SpMV RAM as BRAM limits the parallelism of multiple threads. Upon accessing the BRAM, either the threads stall or the workloads are rescheduled.

A notable point is that the hazard elimination strategies (preventing hazards from happening, ultimately



Supplementary Figure 4: SpMV intrinsic hazards: (a) Intra-stage hazards can occur at the same pipeline stage of different threads. Here, two threads are fetching and later accumulating the same result vector  $R[0]$ . (b) Inter-stage hazards can happen at contiguous pipeline stages of different threads. Here,  $R[0]$  is still accumulated by thread 1, and the old value is fetched again by thread 2. In (a) and (b) next to the thread execution is the COO tuple working queue, which can have arbitrary size and order of elements as they come from the original COO array in the top left of the figure. (c) Structural hazards can happen when multiple threads try to fetch the same vector at the same time.

accelerating the whole system, and avoiding wrong results) of integer SpMV and floating-point SpMV vary. For floating point SpMV, multi-cycle Digital Signal Processors (DSPs) are used, and the respective optimizing strategies usually contain dynamic compression format selection and workload rescheduling. Reference<sup>28</sup> introduces a rescheduling unit to prevent COO elements with the same *row* from being streamed consecutively or simultaneously, normally replacing hazard elements with elements coming later. However, if NNZs are centralized in a few rows, the thread must still stall as there is no element to replace. Some works<sup>29</sup> utilize the CSR format and completely avoid intra-stage hazards. This approach introduces a complex adder tree structure and interconnection to handle the CSR format. Unlike floating point SpMV, in our design hazards are completely solved by using techniques mentioned in Supplementary Note 3.

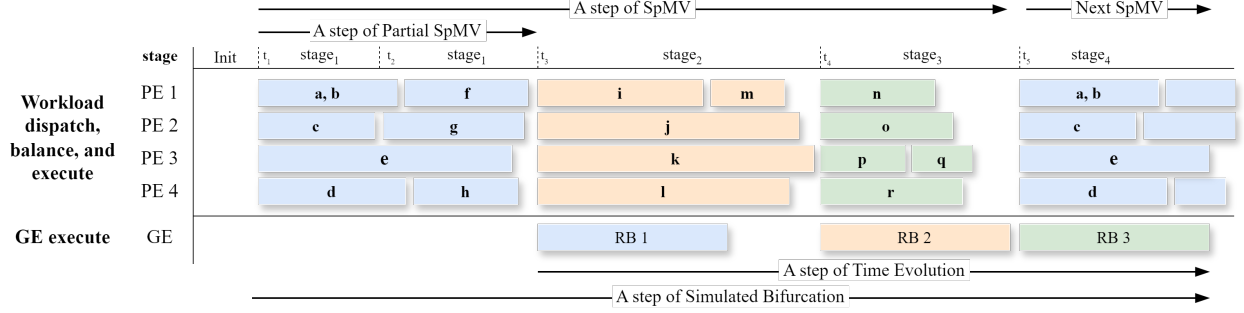
### Supplementary Note 3. Hardware architecture

Supplementary Figure 5 incorporates the overall design of our Ising simulation system. a Dispatcher (Supplementary Figure 5(a)) moves data to and from off-chip memory, also ensuring a coordinated execution for Ising simulation. There are four PEs to perform SpMV operations (one PE is shown in the Supplementary Figure 5(b), the others are stacked behind), a Gather Element (GE, (Supplementary Figure 5(c))) is assigned to perform the time evolution (TE), encompassing all operations outside of SpMV, a *ping pong* SpMV RAM (Supplementary Figure 5(f)) that stores the SpMV results from PEs for subsequent GE processing, a *ping pong vector* RAM ((Supplementary Figure 5(g))) that stores current and new Ising variable  $x$  and  $y$ .

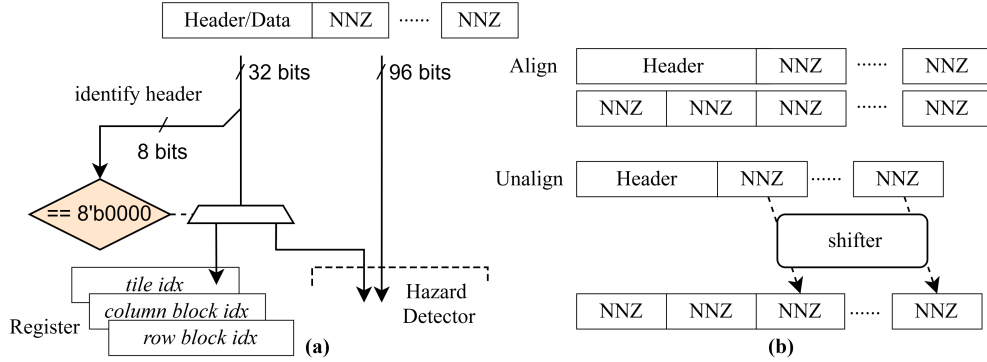
The Ising simulation begins with the Dispatcher configuring the qSB parameters and initializing the bifurcation variables  $x$  and  $y$  by loading a randomly generated vector into the on-chip *ping-pong vector RAM* (Supplementary Figure 5(g)). During each qSB iteration, the Dispatcher fetches scheduled TCOO data from off-chip memory and distributes it to the PEs. These PEs perform column-block-level SpMV upon receiving the *PE non-zero stream* (Supplementary Figure 5(b)).

When processing a new column block, each PE first generates a vector request through the *vector request queue* (Supplementary Figure 5(b)), prompting the Dispatcher to update the PE's local vector RAM via the *vector stream*. The PE then computes the SpMV for each non-zero element encoded in the COO tuples. Following completion of a row-block partial SpMV across all PEs, the resulting vector is transferred to the GE through the *ping-pong* SpMV RAM for TE. During this phase, the PEs concurrently begin processing the next row block. The GE executes TE by reading the vector from the *ping-pong* SpMV RAM and writing





Supplementary Figure 6: Example Ising simulation flow for the matrix example from the Main Text. Where 4 PEs receive balanced workloads from DRAM. The upper timeline explains the lifetime of each SpMV/partial SpMV operation. The lower timeline explains the lifetime of Ising simulation process. Here,  $RB_i$  denotes the SpMV result of the respective row block that GE performs TE on,  $PE_j$  is the  $j$ -th PE, and  $stage_k$  is the maximum execution period of the current row block SpMV or last row block TE, indexed by  $k$ .

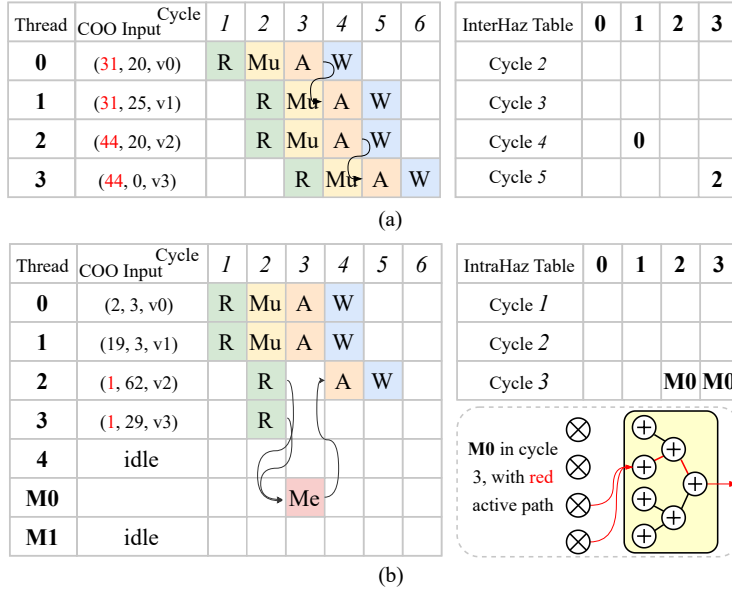


Supplementary Figure 7: (a) Header identification at the beginning of PE; The first 32 bits (header width) will not be sent to Hazard Detector if PE reads an 8-bit zero, but will be used to update tile, column block, and row block address registers indicating that PE is processing the tiles at that address. The rest of 6 non-zeros (96 bits) will be directly sent into Hazard Detector to resolve the hazard. (b) If we use another size zero (no 8-bit), the data next to the header (6 NNZs) will be malposition to the second data packet. A shifter is needed to re-align data to the receiving register by moving the bits using combinational logic.

that  $e_i.row$  refers to the row value of  $e_i$  of a certain NNZ (and analogously for the other values). We use  $v[j]$  to represent the  $j$ -th value of the multiplied vector. And we use  $r_{i,old}[k]$  to represent the last calculated  $k$ -th value of the result vector in the pipeline,  $r_{i,RAM}[k]$  to represent the last calculated  $k$ -th value of the result vector in the RAM, and  $r_{i,new}[k]$  to represent the newest calculated  $k$ -th value of the result vector in the pipeline. During each cycle, the PE first identifies if there is a TCOO header in the data stream, as in Supplementary Figure 7. If there is a header, it updates the tile, column block, and row block address register and removes the header. The rest of the data will be sent to the Hazard Detector. If there is no header, all data will be sent to Hazard Detector. The calculation on the PE runs until the last column block. After processing, the calculation proceeds with the next row block.

To address the SpMV data dependency problem (Supplementary Note 2), several strategies are adopted in Hazard Detector and ALU to overcome the SpMV hazards.

1. **Data Forwarding** (Supplementary Figure 8(a)). This allows data from each PE thread to be forwarded to all other threads to prevent read-after-write (RAW) hazards by identifying and recording conflicting NNZ row between threads at the *Read* and *Accumulate* stages.



Supplementary Figure 8: Hazard elimination: (a) Inter-stage hazard detected between thread 0 and 1 and thread 2 and 3, and the accumulated results of thread 0 and 2 are forwarded to thread 1 and 3. Here, R: read; Mu: multiply; A: accumulate; W: write back. The values of the COO tuple is not emphasized here and represented by  $v_0, \dots, v_3$ . The right side gives inter-hazard table value in each cycle: In cycle 4, the result vector value of thread 0 should be forwarded to thread 1, In cycle 5, the result of thread 2 should be forwarded to thread 3. (b) Intra-stage hazard detected between thread 2 and 3, the multiplication results of thread 2 and 3 are merged by M0 (here, Me: merge). The right side gives intra-hazard table merge adder assignment in each cycle and merge adder active path for M0 (M2 and M3 are hidden for simplicity as they are inactive).

2. **Merge Addition** (Supplementary Figure 8(b)). This inputs NNZs of eight threads with identical *row* to the merge adder in the *Merge* stage, sidestepping intra-stage hazards. A recent FP version design is proposed by work<sup>30</sup>, with a more complex design.
3. **Register-based RAM**. Replacing the Blocked RAM (BRAM) with registers eliminates the structural hazard. The atomic operation of register access is guaranteed by hazard elimination mentioned above, ensuring that any register will not be read or written by two or more threads simultaneously. However, when the register address width increases as  $O(N)$ , FPGA LUT usage increases by  $O(2^N)$ . We carefully implement the register address range to avoid any logic overhead.

**Hazard Detector.** In the Hazard Detector, after receiving the eight NNZs, a RAW hazard potentially occurs between the *Accumulate* stage of any thread and the *WriteBack* stage of any thread. Hence, between the *Read* and *Accumulate* stages,  $N_{\text{thread}}^2$  comparisons are made (where  $N_{\text{thread}}$  is the number of threads) to identify RAW conflicts by comparing the row between the *Read* and *Accumulate* stages of all threads. The positions of NNZs with the same row index are recorded in the inter-hazard table (Supplementary Algorithm 3, 1-4). For example, in Supplementary Figure 8(a), the result of the vector  $\mathbf{v}_0$  in the *Write* stage of thread 0 is identified and forwarded to the *Accumulate* stage of thread 2.

Moreover, to address the intra-stage hazard, the row of each NNZ is read and compared with the row of the remaining seven NNZs. If the same row is identified, the NNZ with the smaller entry number is

designated as the source of the conflict. All conflicted entry indexes and NNZs are recorded in the intra-hazard table (Supplementary Algorithm 3, 5-15). Upon accessing the table (Supplementary Figure 8(b)), the ALU dispatches the NNZs and vertices to the merge adders (M0 to M3, while M2 and M3 are not depicted for simplicity). The dynamic allocation is determined by the variable number of conflicts in each round. An additional adder tree is utilized to determine which results need to be sent to merge adders, enabling the eight NNZs to be dynamically dispatched to any one of the merge adders (Supplementary Algorithm 3, 16-20). For example, in Supplementary Figure 8(b), during the *Read* stage, thread 2 and thread 3 receive NNZs with the same row 1. Thread 2 is identified as the source of conflict. In the *Accumulate* stage, following multiplication,  $\mathbf{R}[1]$  and  $\mathbf{R}[3]$  are directed to merge the adder M0, as dictated by the intra-hazard table. After addressing the hazard, the Hazard Detector also fetches the vector from the PE vector RAM by NNZ column index as  $\mathbf{v}[e_i.col]$  and the old SpMV value from the SpMV RAM according to the NNZ row  $\mathbf{SpMV}[e_i.row]$ , and sends them to ALU.

```

input: NNZ in read stage:  $\mathbf{e}[]$ 
        NNZ in accumulate stage:  $\mathbf{e\_acc}[]$ 
        multiplied vector:  $\mathbf{v}[512]$ 

output: intra-hazard table : intrahaz_table
        inter-hazard table : interhaz_table

% generate interhaz_table for i ← 0 to 8 do
for j ← 0 to 8 do
    if  $\mathbf{e}[i].row == \mathbf{e\_acc}[j].row$  and  $\mathbf{NNZ}[i]$  and  $\mathbf{e\_acc}[j]$  is valid then
        interhaz_table[i] = j

% generate intrahaz_table for i ← 0 to 8 do
for j ← i + 1 to 8 do
    if  $\mathbf{e}[i].row == \mathbf{e}[j].row$  and  $\mathbf{e}[i]$  and  $\mathbf{e}[j]$  are valid then
        if intrahaz_table[i] not True then
            intrahaz_table[i].valid = True intrahaz_table[i].row =  $\mathbf{NNZ}[i].row$  intrahaz_table[i].e.append( $\mathbf{e}[i]$ ,  $\mathbf{v}[\mathbf{e}[i].col]$ ) intrahaz_table[i].e.append( $\mathbf{e}[j]$ ,  $\mathbf{v}[\mathbf{e}[j].col]$ ) invalidate  $\mathbf{e}[j]$ 
        else
            intrahaz_table.append( $\mathbf{e}[j]$ ,  $\mathbf{v}[\mathbf{e}[j].col]$ ) invalidate  $\mathbf{e}[j]$ 

% assign merge adder i = 0 for i ← 0 to 8 do
if intrahaz_table.valid then
    intrahaz_table.adder = i i ++

```

Supplementary Algorithm 3: Hazard Detector

**ALU.** The ALU comprises the stages *Multiplication*, *Merge*, and *Accumulate*. In the *multiplication* stage, NNZs multiply with the vector, as

$$mm_i = e_i * \mathbf{v}[e_i.col].$$

In the *Merge* phase, the Intra-hazard Table is read, and all NNZs sharing the same source ( $e.row$ ) are input into the merge adder where the NNZs are accumulated as

$$mm_i = mm_{i+1} + \dots + mm_{i+n}.$$

Given that intra-stage hazard appears at most  $N_{\text{thread}}/2$  times in each cycle, we apply four additional 8-to-1 adder arrays. After merging, the decision to forward the latest result  $r$  is indicated by the Inter-hazard

Table, as shown by the equation,

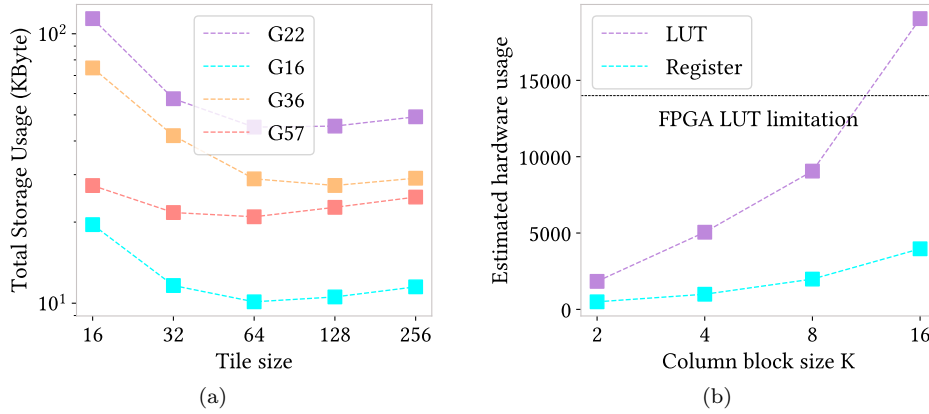
$$r_{i,\text{old}} = \begin{cases} r_{i,\text{new}}[\text{interhaz\_table}[i]] & \text{if } i \text{ has hazard,} \\ r_{i,\text{RAM}} & \text{if } i \text{ has no hazard.} \end{cases}$$

Ultimately, all new SpMV values are added to the original values and written back as

$$r_{i,\text{new}} = r_{i,\text{old}} + \text{mm}_i.$$

**Gather Elements.** Upon completion of the PE calculations for a row block, the ping pong SpMV RAM A will be switched to GE and SpMV RAM B will be connected to PE. Therefore, PE and GE calculations can overlap. During switching, the corresponding vector is streamed to GE vector RAM to prevent the GE from idling. Both will be used by the GE to perform the TE. The quantized  $\mathbf{ps}[i]$ ,  $\xi$ , and decreased  $\Delta t$  (mentioned in Supplementary Algorithm 2) are set up by Dispatcher. The vector is fetched from the current vector RAM 32 rows at a time. The TE as described in Supplementary Algorithm 2 is proceeded in parallel. After the calculation, the GE directly sends 32 new vertices to the new vector RAM.

## Supplementary Note 4. TCOO benchmark



Supplementary Figure 9: Performance modeling: (a) TCOO compressed graph memory usage on tile size  $M$  from 16 to 256 with different graphs; (b) FPGA resource usage on different column blocks size  $K$  with  $M$  held constant. The dotted line is the LUT resource limitation for the ZCU106 FPGA. When  $K$  is larger than 8, the LUT usage exceeds the limitation.

**Comparison with sparse format.** To analyze the performance of TCOO, we first determined the best TCOO parameters by traversing the parameter space and searching the parameter with the lowest memory usage. Then, we compared the TCOO memory usage of the Gset<sup>31</sup> matrix with other sparse matrix formats (COO, BCOO, CSR, and CSR5<sup>22</sup>). Gset is a large-scale, sparse, and weighted graph set maintained by Stanford University, widely used in combinatorial optimization, graph theory, and quantum-inspired computing research. It includes graphs ranging from 800 to 20,000+ nodes, with varying densities and structures, suitable for optimization problems like Max-Cut, graph partitioning, and Ising model simulations.

As shown in Supplementary Figure 9(a), increasing  $M$  produces a negligible increase in the compression ratio when  $M$  exceeds 64. Additionally, in terms of hardware-software co-optimization, we demonstrate the FPGA resource usage under different column block sizes  $K$  in Supplementary Figure 9(b) (with  $M$  held constant). It is evident that when  $K$  is 16, the resource usage exceeds the total available FPGA resources. Therefore, in our example design, we set  $M$  to 64 and  $K$  to 8.

Next, Supplementary Table 3 compares TCOO memory usage with other formats. We pick BCOO and CSR5 as both formats are designed for varying matrix sizes just as TCOO, which saves redundant address bits when the problem size is not large enough. However, COO and CSR cannot adapt to varying matrix sizes and their address bits are aligned to the max matrix size (20000).

Selecting an appropriate tile size  $M$  is critical to balance header overhead against the advantages of tiling. We extended our evaluation to include tile sizes of  $M = 128$  and  $M = 256$  to account for varying levels of graph density. In most instances, the TCOO format with  $M = 64$  or  $M = 128$  exhibits the lowest memory consumption. Beyond a certain matrix size, however, the overhead from additional headers outweighs the benefits of finer tiling. Hence, for larger matrices, TCOO demonstrates improved efficiency with increasing  $M$ , as its memory footprint comprises both header overhead and non-zero data storage.

We observe that in specific cases (e.g., graphs G55, G56, G60, G61, G70), CSR5 outperforms TCOO, achieving an average memory reduction of approximately 11.3%. This advantage comes from the design efficacy of CSR5 for highly irregular matrices, particularly those with non-zero elements concentrated along diagonal bands separated by sparse regions. Such matrices exhibit a pronounced sparsity imbalance, with the left regions significantly denser than the right regions, as shown in Supplementary Figure 10(a). Although finer tiles would be beneficial in sparse regions, TCOO currently does not support dynamic tile-size adjustment. Moreover, as illustrated in Supplementary Figure 10(b), only tiles fully aligned with diagonal bands achieve high storage efficiency. Tiles near the edges of these bands contain few non-zero entries, resulting in disproportionately high header overhead and thus diminishing the benefits of tiling.

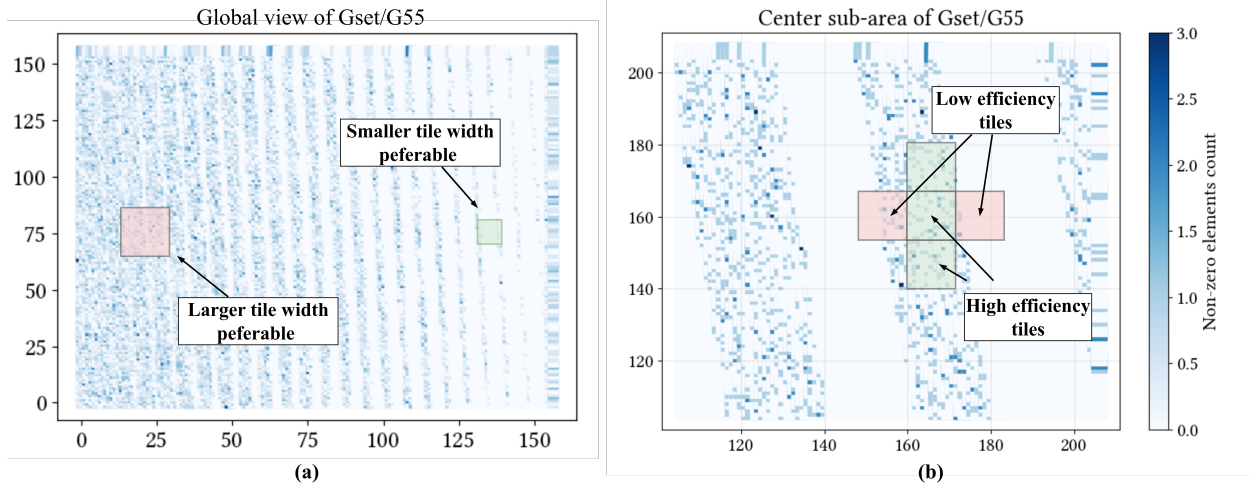
Supplementary Table 3: Gset memory usage comparison; COO, BCOO, CSR, and CSR5 are compared with TCOO; In most of the cases, TCOO has a leading advantage (the best result is highlighted)

| Instance | $N$  | Memory Usage (KB) |        |       |       |              |           |           |
|----------|------|-------------------|--------|-------|-------|--------------|-----------|-----------|
|          |      | COO               | BCOO   | CSR   | CSR5  | TCOO         |           |           |
|          |      |                   |        |       |       | $M = 64$     | $M = 128$ | $M = 256$ |
| G1       | 800  | 79.59             | 127.76 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G2       | 800  | 79.59             | 127.91 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G3       | 800  | 79.59             | 128.50 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G4       | 800  | 79.59             | 128.15 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G5       | 800  | 79.59             | 128.70 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G6       | 800  | 79.59             | 127.76 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G7       | 800  | 79.59             | 127.91 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G8       | 800  | 79.59             | 128.50 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G9       | 800  | 79.59             | 128.15 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G10      | 800  | 79.59             | 128.70 | 81.10 | 49.74 | <b>38.44</b> | 42.42     | 46.91     |
| G11      | 800  | 6.64              | 7.32   | 40.33 | 4.15  | <b>3.35</b>  | 3.64      | 3.98      |
| G12      | 800  | 6.64              | 7.32   | 40.33 | 4.15  | <b>3.35</b>  | 3.64      | 3.98      |
| G13      | 800  | 6.64              | 7.32   | 40.33 | 4.15  | <b>3.35</b>  | 3.64      | 3.98      |
| G14      | 800  | 19.48             | 32.26  | 47.51 | 12.18 | <b>10.16</b> | 10.60     | 11.55     |
| G15      | 800  | 19.34             | 31.88  | 47.43 | 12.09 | <b>10.09</b> | 10.53     | 11.47     |
| G16      | 800  | 19.39             | 32.01  | 47.46 | 12.12 | <b>10.12</b> | 10.55     | 11.50     |
| G17      | 800  | 19.37             | 31.96  | 47.45 | 12.10 | <b>10.11</b> | 10.54     | 11.49     |
| G18      | 800  | 19.48             | 32.26  | 47.51 | 12.18 | <b>10.16</b> | 10.60     | 11.55     |
| G19      | 800  | 19.34             | 31.88  | 47.43 | 12.09 | <b>10.09</b> | 10.53     | 11.47     |
| G20      | 800  | 19.39             | 32.01  | 47.46 | 12.12 | <b>10.12</b> | 10.55     | 11.50     |
| G21      | 800  | 19.37             | 31.96  | 47.45 | 12.10 | <b>10.11</b> | 10.54     | 11.49     |
| G22      | 2000 | 82.97             | 144.07 | 82.98 | 51.85 | <b>45.04</b> | 45.42     | 49.18     |

Continued on next page

Table 3 – Continued

| Instance | $N$   | COO    | BCOO   | CSR    | CSR5         | TCOO         |              |               |
|----------|-------|--------|--------|--------|--------------|--------------|--------------|---------------|
|          |       |        |        |        |              | $M = 64$     | $M = 128$    | $M = 256$     |
| G23      | 2000  | 82.97  | 144.39 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G24      | 2000  | 82.97  | 144.45 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G25      | 2000  | 82.97  | 144.31 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G26      | 2000  | 82.97  | 144.17 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G27      | 2000  | 82.97  | 144.07 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G28      | 2000  | 82.97  | 144.27 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G29      | 2000  | 82.97  | 144.45 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G30      | 2000  | 82.97  | 144.31 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G31      | 2000  | 82.97  | 144.17 | 82.98  | 51.85        | <b>45.04</b> | 45.42        | 49.18         |
| G32      | 2000  | 16.60  | 18.31  | 45.90  | 10.38        | <b>8.39</b>  | 9.07         | 9.91          |
| G33      | 2000  | 16.60  | 25.20  | 45.90  | 10.38        | <b>8.39</b>  | 9.07         | 9.91          |
| G34      | 2000  | 16.60  | 18.31  | 45.90  | 10.38        | <b>8.39</b>  | 9.07         | 9.91          |
| G35      | 2000  | 48.88  | 83.36  | 63.94  | 30.55        | 28.92        | <b>27.38</b> | 29.13         |
| G36      | 2000  | 48.83  | 83.00  | 63.91  | 30.52        | 28.94        | <b>27.35</b> | 29.10         |
| G37      | 2000  | 48.91  | 83.35  | 63.95  | 30.57        | 28.96        | <b>27.39</b> | 29.15         |
| G38      | 2000  | 48.89  | 83.17  | 63.94  | 30.55        | 28.94        | <b>27.38</b> | 29.13         |
| G39      | 2000  | 48.88  | 83.36  | 63.94  | 30.55        | 28.92        | <b>27.38</b> | 29.13         |
| G40      | 2000  | 48.83  | 83.00  | 63.91  | 30.52        | 28.94        | <b>27.35</b> | 29.10         |
| G41      | 2000  | 48.91  | 83.35  | 63.95  | 30.57        | 28.96        | <b>27.39</b> | 29.15         |
| G42      | 2000  | 48.89  | 83.17  | 63.94  | 30.55        | 28.94        | <b>27.38</b> | 29.13         |
| G43      | 1000  | 41.46  | 70.97  | 59.79  | 25.91        | <b>21.01</b> | 22.33        | 24.48         |
| G44      | 1000  | 41.46  | 70.96  | 59.79  | 25.91        | <b>21.01</b> | 22.33        | 24.48         |
| G45      | 1000  | 41.46  | 70.84  | 59.79  | 25.91        | <b>21.01</b> | 22.33        | 24.48         |
| G46      | 1000  | 41.46  | 71.11  | 59.79  | 25.91        | <b>21.01</b> | 22.33        | 24.48         |
| G47      | 1000  | 41.46  | 70.84  | 59.79  | 25.91        | <b>21.01</b> | 22.33        | 24.48         |
| G48      | 3000  | 24.90  | 27.47  | 50.54  | 15.56        | <b>12.56</b> | 13.62        | 14.86         |
| G49      | 3000  | 24.90  | 27.47  | 50.54  | 15.56        | <b>13.10</b> | 13.62        | 14.86         |
| G50      | 3000  | 24.90  | 27.47  | 50.54  | 15.56        | <b>13.08</b> | 13.62        | 14.86         |
| G55      | 5000  | 51.87  | 91.38  | 65.61  | <b>32.42</b> | 59.67        | 36.64        | 32.86         |
| G56      | 5000  | 51.87  | 91.38  | 65.61  | <b>32.42</b> | 59.67        | 36.64        | 32.86         |
| G57      | 5000  | 41.50  | 45.78  | 59.81  | 25.94        | <b>20.93</b> | 22.69        | 24.77         |
| G58      | 5000  | 122.73 | 213.24 | 105.20 | 76.70        | 93.21        | <b>74.13</b> | 74.54         |
| G59      | 5000  | 122.73 | 213.24 | 105.20 | 76.70        | 93.21        | <b>74.13</b> | 74.54         |
| G60      | 7000  | 71.17  | 125.40 | 76.39  | <b>44.48</b> | 100.12       | 55.40        | 46.46         |
| G61      | 7000  | 71.17  | 125.40 | 76.39  | <b>44.48</b> | 100.12       | 55.40        | 46.46         |
| G62      | 7000  | 58.11  | 64.09  | 69.09  | 36.32        | <b>30.57</b> | 31.73        | 34.67         |
| G63      | 7000  | 172.07 | 299.69 | 132.78 | 107.54       | 148.85       | 108.82       | <b>105.81</b> |
| G64      | 7000  | 172.07 | 299.69 | 132.78 | 107.54       | 148.85       | 108.82       | <b>105.81</b> |
| G65      | 8000  | 66.41  | 73.24  | 73.73  | 41.50        | <b>34.91</b> | 36.28        | 39.64         |
| G66      | 9000  | 74.71  | 82.40  | 78.37  | 46.69        | <b>39.29</b> | 40.81        | 44.59         |
| G67      | 10000 | 83.01  | 91.55  | 83.01  | 51.88        | <b>43.67</b> | 45.35        | 49.54         |
| G70      | 10000 | 41.50  | 73.19  | 59.81  | <b>25.94</b> | 99.10        | 56.42        | 33.58         |
| G72      | 10000 | 83.01  | 91.55  | 83.01  | 51.88        | <b>43.67</b> | 45.35        | 49.54         |
| G77      | 14000 | 116.21 | 128.17 | 101.56 | 72.63        | <b>61.10</b> | 63.47        | 69.33         |
| G81      | 20000 | 166.02 | 183.11 | 129.39 | 103.76       | <b>87.31</b> | 90.66        | 99.06         |

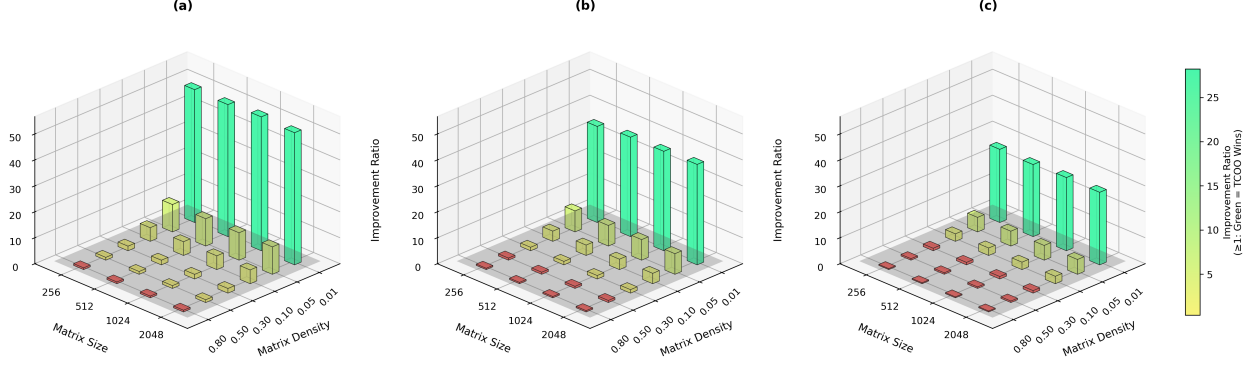


Supplementary Figure 10: TCOO performance analysis on the example of G55. (a) For an imbalanced sparsity matrix, the dense area on the left prefers a larger tile, and the sparse area on the right prefers a smaller tile. (b) The tile cannot accommodate the diagonal distribution very well, making the tiles on the edge of the distribution less efficient.

**Comparison with dense format.** To further validate the efficiency and scalability of the TCOO format, we conducted a comprehensive performance evaluation across multiple dimensions, including matrix density, data type, and matrix size. The improvement ratio is defined as the TCOO compression ratio divided by the best-performing alternative algorithm’s compression ratio, where values greater than 1 indicate TCOO superiority. Our analysis reveals that TCOO consistently outperforms traditional encoding algorithms (Delta Encoding, Bit Packing, and Huffman Coding) across a wide range of sparse matrix scenarios. For INT4 data types at 1% density, TCOO achieves remarkable improvement ratios of 51.20x compared to the best alternative algorithm (Bit Packing), delivering compression ratios up to 412.83x. Similar advantages are observed for INT8 (38.72x improvement) and FP16 (28.21x improvement) data types under the same sparse conditions.

The evaluation demonstrates a clear density-dependent performance characteristic: TCOO maintains significant advantages in sparse regimes (1%-30% density) with improvement ratios ranging from 1.33x to 51.20x across all data types. However, as matrix density increases beyond 50%, Bit Packing becomes more effective for dense matrices, with TCOO showing 0.50x-0.95x improvement ratios in high-density scenarios (50%-80%). This validates that TCOO is particularly optimized for sparse matrix representations while acknowledging the limitations for dense matrices.

Furthermore, the results show consistent performance across different matrix sizes (256x256 to 2048x2048), indicating TCOO’s scalability. The compression efficiency demonstrates minimal variance with increasing matrix dimensions, confirming the format’s robustness for large-scale sparse matrix applications. These quantitative results substantiate TCOO’s claimed efficiency gains for sparse matrix computations while providing clear boundaries for its optimal application scope.



Supplementary Figure 11: Comparison of TCOO compression performance improvement ratios across different data types and matrix characteristics; (a) TCOO compared with other compression algorithms under INT4 data type; (b) TCOO compared with other compression algorithms under INT8 data type; (c) TCOO compared with other compression algorithms under FP16 data type. (improvement ratio  $\geq 1$  gives green and yellow color; improvement ratio  $< 1$  gives red color)

## Supplementary Note 5. SBM benchmark

**Max-cut.** We compare qSBM with four processing elements (**P4**) with dense the dense SBM architecture (**K2000**) and GPU SBM in terms of time-per-step (TPS) when processing the Max-Cut problem on sparse graphs. We add a comparative GPU platform because the most recent SpMV works are mainly done on GPU<sup>7</sup>. The GPU used is NVIDIA RTX4080 (marked as **G**). In GPU and FPGA benchmark, we add time-to-solution (TTS) comparison. Among all state-of-the-art GPU SpMV libraries<sup>32–35</sup>, NVIDIA’s official library cuSPARSE stands out for its universality and high performance. We use cuSPARSE API to perform product-sum operations and time evolution in parallel.

When solving each graph, the temporal parameter  $dt$  is tuned to obtain a better result. Each graph is tested for 100 trials to get the best Max-Cut value. We compared the probability of success to reach the target value  $P_s$  between bSB and qSB to show the performance of **P4**.

As shown in Supplementary Table 4, **P4** achieves consistent speedups over the **G**, with performance gains ranging from  $3.1\times$  (G8) to  $26\times$  (G32). Beyond throughput improvements, the FPGA’s advantages become particularly compelling for real-world deployment scenarios. First, the reconfigurable architecture delivers superior energy efficiency, attributed to the elimination of instruction fetch overhead and precise clock gating for sparse data patterns. Second, the design’s ability to exploit irregular parallelism avoids the GPU’s inherent limitations in warp scheduling and thread divergence. These characteristics position the FPGA as not merely an accelerator, but an energy-proportional solution for NNZ and hybrid computing environments where power constraints and latency requirements preclude GPU deployments.

Besides, **P4** achieves partial speedup compared to **K2000** (G11-G20, G27-G38, and G50). Where **P4** is 10000X faster in the best case (G18) and 1.16X faster in the worst case (G30) than **K2000**. For the graphs where **K2000** performs better, the time for P4 also remains within  $5us$ , with a small gap compared to **K2000**, verifying our conclusion in the main text that our design close the gap between sparse Ising simulation accelerator and the dense one.

To fairly compare architectural efficiency with **K2000**, we focus on instances where both solvers operate effectively. Following the methodology outlined in the Main Text, we excluded extreme outliers (G11, G18, G19, G20) where **K2000**’s exceptionally high time-per-step ( $> 1000us$ ) predominantly reflects algorithmic convergence difficulty on challenging sparse graphs, rather than hardware speed. Among the remaining 45 G-set instances, **P4** achieves a better (lower) time-per-step in 25 cases, with an average speedup of  $5.1\times$ .

This includes significant speedups on many instances (e.g.,  $\sim 38\times$  on G14,  $\sim 71\times$  on G35,  $\sim 135\times$  on G37). For TTS result, on instances where both solvers are effective ( $P_s > 0$ ), **P4**'s significant TPS advantage consistently translates into a TTS speedup. For example: On G14, P4 achieves a  $15.8\times$  TTS speedup over **G**. On G15, the TTS speedup is  $19.1\times$ . Even on instances like G8 where P4's  $P_s$  is lower (46% vs. 78%), its drastically lower TPS (4.5  $\mu$ s vs. 14.0  $\mu$ s) still results in a  $1.7\times$  faster TTS. This analysis quantitatively demonstrates the efficiency of our co-designed sparse architecture for its target problem domain. To note that speedup also stems from sparse data patterns. On the subset of moderately dense graphs where the dense **K2000** design is naturally efficient, **P4**'s performance remains competitive, with its time-per-step consistently under 5 $\mu$ s. This comparison underscores that while a dense solver may excel on regular problems, our sparse design delivers superior and robust performance across the broad spectrum of practical, sparse combinatorial problems.

To provide a comprehensive comparison of solution quality between our approach and classical methods, we conducted a systematic analysis across 49 Gset instances. This analysis reveals that qSB (P4) achieves better cut values in 19 instances (38.8%), while bSB finds superior solutions in 28 instances (57.1%), with ties occurring in 2 instances (4.1%). These results confirm that quantization does not impair solution quality.

Supplementary Table 4: TPSs of **K2000**, **G**, and **P4** for Max-Cut problems on G-set instances are provided together with the following. We use a tuned  $dt$  ranging from 0.25 to 1 to retrieve the best Max-Cut values for **G** and 1 to 20 for **P4** ("-" means that values from the previous work are not provided;  $P_s$  means the success probability for target value)

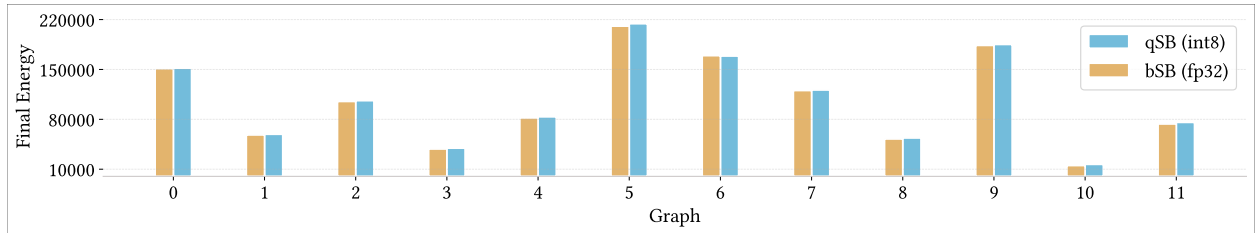
| Instance | $N$  | Best<br>Known | <b>K2000</b> |           | <b>G</b> |         |           | <b>P4</b>  |         |           |
|----------|------|---------------|--------------|-----------|----------|---------|-----------|------------|---------|-----------|
|          |      |               | TPS(us)      | $P_s(\%)$ | TPS(us)  | TTS(ms) | $P_s(\%)$ | TPS(us)    | TTS(ms) | $P_s(\%)$ |
| G1       | 800  | 11624         | 1.0          | 99.0      | 28.1     | 20.9    | 100.0     | 4.5        | 4.2     | 100.0     |
| G2       | 800  | 11620         | 1.0          | 99.0      | 17.2     | 13.4    | 100.0     | 4.5        | 4.3     | 100.0     |
| G3       | 800  | 11622         | 1.0          | 99.0      | 25.7     | 19.6    | 100.0     | 4.5        | 4.2     | 100.0     |
| G4       | 800  | 11646         | 1.0          | 99.0      | 25.8     | 19.2    | 100.0     | 4.5        | 4.2     | 100.0     |
| G5       | 800  | 11631         | 1.0          | 99.0      | 15.2     | 11.7    | 100.0     | 4.5        | 4.3     | 100.0     |
| G6       | 800  | 2178          | 1.1          | 98.9      | 25.7     | 22.1    | 50.0      | 4.5        | 4.3     | 44.0      |
| G7       | 800  | 2006          | 2.5          | 85.5      | 25.7     | 22.3    | 33.0      | 4.5        | 4.3     | 34.0      |
| G8       | 800  | 2005          | 1.0          | 99.0      | 14.0     | 11.8    | 78.0      | 4.5        | 4.3     | 46.0      |
| G9       | 800  | 2054          | 3.6          | 74.2      | 25.7     | 22.4    | 48.0      | 4.5        | 4.3     | 44.0      |
| G10      | 800  | 2000          | 2.4          | 88.1      | 25.7     | 22.2    | 52.0      | 4.5        | 4.3     | 39.0      |
| G11      | 800  | 564           | 2500.0       | 0.2       | 25.7     | -       | 0.0       | <b>0.4</b> | -       | 0.0       |
| G12      | 800  | 556           | 41.3         | 11.4      | 25.7     | -       | 0.0       | <b>0.4</b> | -       | 0.0       |
| G13      | 800  | 582           | 80.8         | 6.0       | 25.8     | -       | 0.0       | <b>0.4</b> | -       | 0.0       |
| G14      | 800  | 3064          | 38.4         | 23.0      | 25.8     | 24.8    | 2.0       | <b>1.0</b> | 1.0     | 1.0       |
| G15      | 800  | 3050          | 42.5         | 21.0      | 25.8     | 25.0    | 41.0      | <b>1.0</b> | 1.0     | 93.0      |
| G16      | 800  | 3052          | 36.4         | 24.1      | 25.8     | 24.4    | 49.0      | <b>1.1</b> | 1.1     | 91.0      |
| G17      | 800  | 3047          | 27.1         | 30.9      | 25.8     | 25.6    | 31.0      | <b>1.1</b> | -       | 1.0       |
| G18      | 800  | 992           | 10016.7      | 0.1       | 32.0     | -       | 0.0       | <b>1.0</b> | -       | 0.0       |
| G19      | 800  | 906           | 3330.0       | 0.3       | 25.8     | -       | 0.0       | <b>1.1</b> | -       | 1.0       |
| G20      | 800  | 941           | 1660.0       | 0.6       | 25.8     | -       | 0.0       | <b>1.1</b> | 1.1     | 2.0       |
| G21      | 800  | 931           | -            | -         | 25.8     | -       | 0.0       | 1.1        | -       | 0.0       |
| G22      | 2000 | 13359         | 1.1          | 98.6      | 29.5     | 27.4    | 69.0      | 4.4        | 4.3     | 100.0     |
| G23      | 2000 | 13344         | 1.0          | 99.0      | 29.9     | 28.2    | 99.0      | 4.4        | 4.3     | 100.0     |
| G24      | 2000 | 13337         | 1.0          | 99.0      | 14.2     | 13.7    | 99.0      | 4.4        | 4.3     | 100.0     |
| G25      | 2000 | 13340         | 1.0          | 99.0      | 26.4     | 25.0    | 97.0      | 4.4        | 4.3     | 100.0     |
| G26      | 2000 | 13328         | 1.0          | 99.0      | 26.0     | 26.0    | 96.0      | 4.4        | 4.3     | 100.0     |

Continued on next page

Table 4 – Continued

| Instance | $N$  | Best<br>Known | K2000   |           | G       |         |           | P4         |         |           |
|----------|------|---------------|---------|-----------|---------|---------|-----------|------------|---------|-----------|
|          |      |               | TPS(us) | $P_s(\%)$ | TPS(us) | TTS(ms) | $P_s(\%)$ | TPS(us)    | TTS(ms) | $P_s(\%)$ |
| G27      | 2000 | 3341          | 7.5     | 48.7      | 26.0    | -       | 0.0       | <b>4.4</b> | -       | 0.3       |
| G28      | 2000 | 3298          | 7.9     | 46.8      | 26.0    | -       | 0.0       | <b>4.4</b> | -       | 0.3       |
| G29      | 2000 | 3405          | 12.0    | 34.0      | 26.2    | -       | 0.0       | <b>4.4</b> | -       | 0.0       |
| G30      | 2000 | 3413          | 5.1     | 62.6      | 26.2    | -       | 0.0       | <b>4.4</b> | 4.3     | 0.7       |
| G31      | 2000 | 3310          | 6.9     | 51.3      | 25.7    | -       | 0.0       | <b>4.4</b> | 4.3     | 0.3       |
| G32      | 2000 | 1410          | -       | -         | 26.0    | -       | 0.0       | 1.0        | -       | 0.0       |
| G33      | 2000 | 1382          | 225.0   | 2.2       | 26.0    | -       | 0.0       | <b>1.0</b> | -       | 0.0       |
| G34      | 2000 | 1384          | 65.0    | 7.4       | 25.9    | -       | 0.0       | <b>1.0</b> | -       | 0.0       |
| G35      | 2000 | 7687          | 191.3   | 5.1       | 25.9    | -       | 1.0       | <b>2.7</b> | -       | 0.0       |
| G36      | 2000 | 7680          | 102.7   | 9.3       | 15.2    | -       | 0.0       | <b>2.7</b> | -       | 0.0       |
| G37      | 2000 | 7691          | 365.7   | 2.7       | 15.3    | -       | 0.0       | <b>2.7</b> | -       | 0.0       |
| G38      | 2000 | 7688          | 188.0   | 5.2       | 25.7    | -       | 0.0       | <b>2.7</b> | -       | 0.0       |
| G39      | 2000 | 2408          | -       | -         | 25.7    | -       | 0.0       | 2.7        | -       | 0.0       |
| G40      | 2000 | 2400          | -       | -         | 25.7    | -       | 0.0       | 2.7        | -       | 0.0       |
| G41      | 2000 | 2405          | -       | -         | 26.0    | -       | 0.0       | 2.7        | -       | 0.0       |
| G42      | 2000 | 2481          | -       | -         | 26.0    | -       | 0.0       | 2.7        | -       | 0.0       |
| G43      | 1000 | 6660          | 1.3     | 98.0      | 26.6    | 26.6    | 100.0     | 2.2        | 2.1     | 100.0     |
| G44      | 1000 | 6650          | 1.0     | 99.0      | 16.9    | 16.9    | 100.0     | 2.2        | 2.1     | 100.0     |
| G45      | 1000 | 6654          | 1.2     | 98.3      | 14.5    | 12.7    | 90.0      | 2.2        | 2.1     | 100.0     |
| G46      | 1000 | 6649          | 1.0     | 99.0      | 26.7    | 23.7    | 100.0     | 2.2        | 2.1     | 100.0     |
| G47      | 1000 | 6657          | 1.0     | 99.0      | 26.6    | 23.3    | 99.0      | 2.2        | 2.1     | 100.0     |
| G48      | 3000 | 6000          | 0.8     | 100.0     | 26.0    | -       | 0.0       | 1.6        | -       | 0.0       |
| G49      | 3000 | 6000          | 0.8     | 99.5      | 26.3    | -       | 0.0       | 1.6        | -       | 0.0       |
| G50      | 3000 | 5880          | 2.6     | 100.0     | 26.5    | -       | 0.0       | <b>1.6</b> | -       | 0.0       |

**Weighted max-cut.** To rigorously evaluate quantization benefits beyond the discrete-weight graphs in the Gset benchmark, we conducted additional experiments on weighted graphs. Concretely, we modified several graphs from the Gset collection (G13, G34, G19, etc.), replaced the edge weights with 4-bit weights. We then compared the accuracy of bSB with qSBM on the 4-bit weighted graphs. As shown in Supplementary Figure 12, qSBM not only maintains competitive performance but actually achieves slightly better Max-Cut values compared to the bSB baseline across multiple graph instances. This demonstrates that the advantages of quantization extend to general weighted graphs and are not limited to the binary weight distributions in the original Gset.



Supplementary Figure 12: Performance comparison between quantized (qSB) and full-precision (bSB) simulated bifurcation on biased Ising graphs. The 8-bit quantized implementation achieves competitive solution quality while offering significant hardware efficiency advantages

Supplementary Table 5: Bmincut result for METIS, bSB, and qSB; Imbalance is calculated as  $|n_1 - n_2|$ . The METIS cut option imbalance tolerance is set to 0%

| Graph      | $\lambda$ | Cut Value |       |       | Imbalance |     |     |
|------------|-----------|-----------|-------|-------|-----------|-----|-----|
|            |           | METIS     | bSB   | qSB   | METIS     | bSB | qSB |
| G6 (800)   | 0.005     |           | -2007 | -1999 |           | 8   | 8   |
|            | 0.01      | -1809     | -2008 | -1976 | 0         | 0   | 2   |
|            | 0.05      |           | -2006 | -1933 |           | 8   | 4   |
| G11 (800)  | 0.005     |           | -490  | -482  |           | 4   | 8   |
|            | 0.01      | -324      | -490  | -478  | 2         | 6   | 2   |
|            | 0.05      |           | -484  | -275  |           | 4   | 10  |
| G35 (2000) | 0.005     |           | 2774  | 2804  |           | 0   | 4   |
|            | 0.01      | 3133      | 2798  | 2840  | 0         | 4   | 8   |
|            | 0.05      |           | 2797  | 2891  |           | 10  | 8   |

**Min-cut with balance term.** To further validate the generality of our approach, we also evaluated its performance on constrained optimization problems with strong penalty terms. Specifically, we formulated the balanced minimum cut (bmincut) problem (q=2), which introduces a quadratic penalty term to enforce equal-sized partitions. The Hamiltonian is defined as:

$$H_{\text{bmincut}}(\mathbf{x}) = \frac{1}{4} \sum_{i,j} J_{ij}(1 - x_i x_j) + \lambda \left( \sum_i x_i \right)^2,$$

where  $x_i \in \{-1, +1\}$ ,  $J_{ij}$  represents the connection weight between nodes  $i$  and  $j$ , and  $\lambda > 0$  controls the strength of the balance constraint. The second term serves as a strong global penalty that enforces  $\sum_i x_i \approx 0$ , making this a more challenging optimization problem compared to unconstrained Max-Cut.

We evaluated both bSB and qSB for solving the bmincut problem on multiple Gset instances with a range of balance strengths. For reference, we also include results from METIS, a state-of-the-art graph partitioning library configured with a 0% imbalance tolerance. The results are summarized in Supplementary Table 5. Both bSB and qSB significantly outperform METIS. For instance, on graph G35, bsb cut value (2774) is 11.4% better than METIS (3133) with imbalance=4, and qSB cut value (2804) is 10.5% better than METIS with imbalance=4. Furthermore, qSB demonstrates performance nearly equivalent to classical baseline bSB, not only in solution quality but also in balance. For instance, on graph G6 qSB achieves a cut value of -1999, differing from bSB’s -2007 by only 0.4%, while producing an identical imbalance of 8. On graph G11, qSB not only matches the cut quality but also yields a more balanced partition (imbalance of 2 versus 6 for bSB). On the larger instance G35, the cut values for bSB (2774) and qSB (2804) differ slightly at 1.1%, with both maintaining excellent balance (0 and 4 imbalance, respectively). This indicates that qSB effectively handles balance constraints while producing cut quality nearly matching that of its classical counterpart.

**Qplib.** To further validate the general-purpose optimization capability of our approach, we evaluate it on 23 QPN (unconstrained quadratic binary optimization problems) instances from the QPLIB library, a comprehensive collection of quadratic programming problems spanning diverse application domains, including combinatorial optimization, scheduling, and network design. Unlike the sparse Max-Cut benchmarks, these QPLIB instances include mixed quadratic-linear objectives with both bilinear terms and linear coefficients, indefinite Hessians characterized by both positive and negative eigenvalues, objective values ranging from hundreds to tens of millions, and real-world problem structures that test the robustness of any optimization method.

Supplementary Table 6: QPlib QPN problem result for bSB and qSB. The Diff measure the difference between qSB and best known result.

| Problem    | Vars | Sense | Best Known  | bSB Score   | qSB Score   | Diff |
|------------|------|-------|-------------|-------------|-------------|------|
| QPLIB_3506 | 496  | Max   | 478.00      | 478.00      | 478.00      | 0.00 |
| QPLIB_3565 | 276  | Max   | 282.00      | 280.00      | 282.00      | 0.00 |
| QPLIB_3642 | 1035 | Max   | 1034.00     | 1022.00     | 1028.00     | 0.58 |
| QPLIB_3650 | 946  | Max   | 922.00      | 914.00      | 916.00      | 0.65 |
| QPLIB_3693 | 1128 | Max   | 1154.00     | 1138.00     | 1144.00     | 0.87 |
| QPLIB_3705 | 378  | Max   | 384.00      | 384.00      | 384.00      | 0.00 |
| QPLIB_3706 | 703  | Max   | 682.00      | 680.00      | 680.00      | 0.29 |
| QPLIB_3738 | 435  | Max   | 422.00      | 416.00      | 420.00      | 0.47 |
| QPLIB_3745 | 325  | Max   | 334.00      | 332.00      | 332.00      | 0.60 |
| QPLIB_3822 | 861  | Max   | 850.00      | 846.00      | 848.00      | 0.24 |
| QPLIB_3832 | 561  | Max   | 554.00      | 548.00      | 548.00      | 1.08 |
| QPLIB_3838 | 780  | Max   | 746.00      | 734.00      | 738.00      | 1.07 |
| QPLIB_3850 | 1225 | Max   | 1198.00     | 1182.00     | 1184.00     | 1.17 |
| QPLIB_3852 | 231  | Max   | 234.00      | 234.00      | 234.00      | 0.00 |
| QPLIB_3877 | 630  | Max   | 602.00      | 600.00      | 600.00      | 0.33 |
| QPLIB_5721 | 300  | Max   | 8579363.00  | 8462391.00  | 8485111.00  | 1.10 |
| QPLIB_5725 | 343  | Max   | 33611981.00 | 33432681.00 | 33335147.00 | 0.82 |
| QPLIB_5755 | 400  | Max   | 24838942.00 | 24621533.00 | 24589573.00 | 1.00 |
| QPLIB_5875 | 200  | Max   | 43757.00    | 43396.00    | 43396.00    | 0.83 |
| QPLIB_5881 | 120  | Max   | 13067.00    | 12866.00    | 13053.00    | 0.11 |
| QPLIB_5882 | 150  | Max   | 25388.00    | 25202.00    | 25231.00    | 0.62 |
| QPLIB_5909 | 250  | Max   | 35726.00    | 34863.00    | 35331.00    | 1.11 |
| QPLIB_5922 | 500  | Max   | 128339.00   | 127656.00   | 127895.00   | 0.37 |

Each QPLIB instance we consider is a binary quadratic program of the form

$$\underset{\mathbf{x} \in \{0,1\}^N}{\text{maximize}} \quad \frac{1}{2} \mathbf{x}^\top Q \mathbf{x} + \mathbf{b}^\top \mathbf{x} + c,$$

where  $Q \in \mathbb{R}^{N \times N}$  and  $\mathbf{b} \in \mathbb{R}^N$ . To apply our bifurcation solvers

We apply both bSB and qSB variants with multiple parameter settings, conducting grid searches over  $\Delta t$  values to identify optimal configurations for each problem. As shown in the Supplementary Table 6, across all 23 instances, our method consistently produces high-quality solutions within 0% to 1.17% of the best-known values, with 18 problems achieving the solution  $< 1\%$  of the ground state and 4 problems reaching the ground state. These results support qSB as a robust method towards general QUBO problems with constraints and diverse problem structures.

**Disentangling Quantization Effects.** To systematically investigate the performance of quantized Ising machines, we conducted a controlled ablation study that disentangles the effects of quantization from algorithmic parameter tuning. Three configurations were tested on the Gset benchmark graphs to isolate the contributions of each component: Configuration A (FP32): Full-precision ballistic simulated bifurcation with the optimal FP32  $\Delta t$ , Configuration B (INT8, FP32  $\Delta t$ ): 8-bit quantized bSB while strictly maintaining the identical  $\Delta t$  step size from Config A, and Configuration C (INT8, tuned  $\Delta t$ ): The complete qSB implementation featuring both 8-bit quantization and a  $\Delta t$  value optimized for the quantized dynamics.

The results, summarized in Supplementary Table 7, reveal distinct and interpretable contributions. Performance is evaluated using the average percentage deviation from the best-known solution (Avg) and the

variance across runs (Var), indicating solution quality and stability, respectively. First, quantization alone (Config B) causes significant degradation: solution quality drops substantially (e.g., Avg rises from 5% to over 66% on several instances) and variance increases dramatically (e.g., from tens to thousands). This demonstrates that reduced precision coarsens the state-update granularity, impairing the solver’s ability to navigate the energy landscape effectively and leading to unstable, suboptimal dynamics.

Critically, the full qSB configuration (Config C) demonstrates that re-exploring the parameter space—specifically, tuning  $\Delta t$  for the quantized system—not only recovers but can surpass the full-precision baseline. It achieves a lower average deviation (e.g., 1–4% Avg on most graphs) and significantly reduces variance. On certain challenging instances (e.g., G47), qSB (Config C) achieves *superior* performance (0.36% Avg) compared to the full-precision baseline (0.47% Avg), while maintaining dramatically lower variance (5.93 vs. 140). This counterintuitive result suggests that for some problem classes, the quantization process itself may introduce a beneficial form of stochasticity or “implicit annealing” that enhances exploration capability. This confirms that optimized quantization parameters restore and enhance effective, stable solver dynamics.

Importantly, this recovery is not achieved by trading computational efficiency for accuracy. All experiments were conducted with an identical number of total steps. Thus, qSB does not compensate for precision loss by simply “slowing down” the schedule. Instead, with a similar computational budget, it achieves superior search dynamics. The underlying reason is that the inherent nonlinearities introduced by quantization—such as dynamic clipping and variable binarization—naturally align with the intrinsic nonlinear processes of the bSB algorithm (e.g., its ballistic term). When properly calibrated, quantization does not merely compress the model; it introduces a form of dynamics that synergistically interacts with the algorithm’s core mechanics, leading to more stable and performant solving.

Supplementary Table 7: Quantization Performance Analysis

|   |        | G13  | G34  | G19  | G21   | G20  | G18  | G51  | G53  | G54  | G47  | G40  | G39  | G42  | G41  |
|---|--------|------|------|------|-------|------|------|------|------|------|------|------|------|------|------|
| A | Avg(%) | 4.84 | 7.77 | 3.62 | 2.16  | 2.31 | 2.86 | 2.36 | 2.33 | 2.46 | 0.47 | 4.70 | 2.91 | 4.16 | 5.75 |
|   | Var    | 26.3 | 150  | 38.5 | 36.11 | 131  | 6.61 | 107  | 104  | 98.9 | 140  | 31.8 | 60.0 | 110  | 171  |
| B | Avg(%) | 39.2 | 43.7 | 50.8 | 50.0  | 48.4 | 45.5 | 9.87 | 9.83 | 9.68 | 13.1 | 41.9 | 40.7 | 40.3 | 40.9 |
|   | Var    | 271  | 534  | 883  | 632   | 806  | 728  | 389  | 436  | 348  | 2634 | 1986 | 1399 | 1570 | 1944 |
| C | Avg(%) | 4.63 | 5.01 | 3.83 | 3.65  | 3.58 | 4.04 | 1.00 | 1.12 | 1.46 | 0.36 | 4.18 | 3.76 | 4.15 | 4.63 |
|   | Var    | 29.5 | 71.6 | 127  | 131   | 177  | 166  | 47.6 | 0.36 | 7.10 | 5.93 | 309  | 292  | 301  | 383  |

**Qualitative comparison of Ising machine approaches** Recent works have provided specific physical platforms for Ising optimization, such as the superparamagnetic Ising annealer<sup>36</sup>, fully-connected probabilistic computing accelerator<sup>37</sup>, RRAM-based compute-in-memory Ising machine<sup>38</sup>, programmable optoelectronic Ising machine<sup>39</sup>, and the ferroelectric compute-in-memory annealer<sup>40</sup>. These architectures excel in their respective domains—e.g., leveraging intrinsic stochasticity of superparamagnetic tunnel junctions (SMTJs) for analog annealing, employing compressed formats in RRAM for efficient matrix processing, and using optoelectronic oscillators for high-speed solutions. However, they are often constrained by their specialized platforms: SMTJ and optical systems are typically tailored for dense or compressible all-to-all graphs, RRAM/FeFET-CIM requires custom memory integration, and analog systems generally face challenges with precision, reproducibility, and pre-computation calibration overhead. In contrast, our FPGA-based approach is based on digital hardware, which offers large flexibility, stability, and reproducibility without compromising solution speed, thereby remaining less constrained by platform-specific limitations.

Supplementary Table 8: qSBM FPGA implementation resource comparison; In the BRAM resource, we differentiate vector BRAM (bold) and other BRAM

| Design          | <b>P2</b>           | <b>P4</b>           | <b>MA</b>     | <b>K2000</b>           | <b>K4000</b> |
|-----------------|---------------------|---------------------|---------------|------------------------|--------------|
| Platform        | Xilinx ZCU106       |                     | Xilinx ZCU106 | Intel Arria 10 GX 1150 |              |
| Flexibility     | High                | High                | High          | Low                    | Low          |
| LUTs            | 47314 (21%)         | 107106 (46%)        | –             | 260953                 | 256749       |
| Registers       | 30629 (7%)          | 60316 (13%)         | –             | 281274                 | 288556       |
| BRAM            | 15+ <b>24</b> (12%) | 58+ <b>48</b> (34%) | –             | 756                    | 1524         |
| DSPs            | 66 (4%)             | 130 (8%)            | –             | 104                    | 104          |
| Power (W)       | 5.109               | 4.635               | –             | 48.7                   | 51           |
| Frequency (MHz) | 300                 | 290                 | 150           | 279                    | 269          |
| Efficiency      | 95.0%               |                     | 66%           | 82%                    | 92%          |

## Supplementary Note 6. FPGA benchmark

**Hardware usage comparison.** We present a systematic comparison of our reconfigurable Ising processing architecture against existing implementations (the state-of-the-art sparse FPGA SBM<sup>28</sup> marked as **MA** and **K2000/K4000** in Supplementary Table 8), evaluating three critical metrics: operational flexibility, hardware usage, and computational performance. Our TCOO approach shows substantial advantages on all metrics over conventional dense-matrix implementations as described in the following.

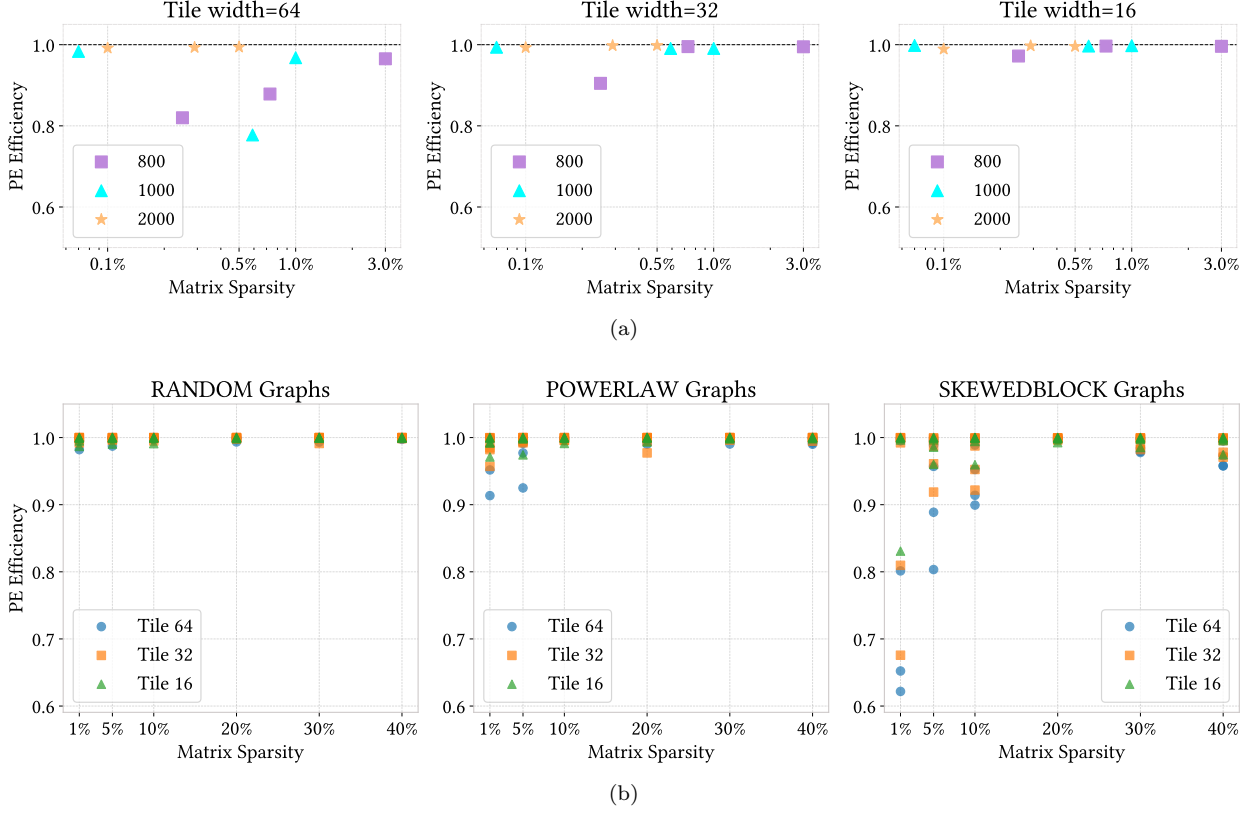
The architectural flexibility, defined as the ability to accommodate varying Ising problem dimensions, proves particularly crucial for practical applications. While the **K2000/K4000** designs suffer severe efficiency degradation when processing non-aligned matrix sizes with up to 75% resource idling in worst-case scenarios, our design maintains consistent performance through adaptive resource allocation. This adaptability stems from our tile-based processing paradigm, which dynamically adjusts to problem characteristics without hardware reconfiguration.

At the hardware usage level, we achieve several key breakthroughs in basic FPGA primitives: (1) LUT6 blocks implementing critical data path logic, which implement any 6-input and 1 output logic function, (2) cascaded BRAM modules providing efficient data storage, and (3) DSP slices accelerating core fixed-point multiplication operations. First, the implementation demonstrates nearly double the operational frequency (300 MHz) compared to the **MA** (150 MHz), while simultaneously improving computational efficiency from 66% to 95%. Second, through careful co-design of the memory hierarchy and processing elements, we reduce BRAM utilization by 13X relative to the **K2000/K4000** implementations. These improvements originate from our novel sparse matrix compression scheme, which reduces memory footprint without compromising data accessibility.

We also list a detailed module-level resource breakdown in Supplementary Table 9 from the Vivado hierarchical utilization report, which reports the utilization of LUTs, registers, BRAMs, and DSPs for the main components (PE, Dispatcher, GE, and Vector RAM). The results show that the PE module consumes the largest share of LUTs (70.0%), followed by the GE (3.5%), Vector RAM (0.5%), and Dispatcher (0.3%).

We have analyzed power consumption across different sparsity levels. Due to our fixed-throughput architecture, the switching activity in the datapath only depends on data structure. Consequently, dynamic power shows no significant variation with sparsity, confirming that power consumption is independent of the input data sparsity in our design.

**PE load balancing.** Load balancing analysis reveals that while our bin-packing distribution algorithm maintains 95% PE efficiency for most problem instances, certain pathological cases—particularly matrices



Supplementary Figure 13: Performance modeling of PE efficiency across different graph datasets: (a) PE efficiency distribution across Gset benchmark graphs; (b) PE efficiency as a function of matrix sparsity ( $10^{-4}$  to  $10^{-1}$ ) for synthetic graphs with sizes from 256 to 8,192 nodes.

with concentrated non-zero elements in specific column blocks—can temporarily reduce efficiency. Our investigation identified that in these outliers (blue triangles in Supplementary Figure 13(a)), the fixed tile configuration leads to workload concentration where primarily 2 out of 4 PEs remain actively engaged in computation, while the other PEs experience only minimal activity levels. This imbalance results in an overall efficiency of approximately 75% during these specific computational phases.

However, the TCOO architecture provides inherent flexibility to address such imbalances through parameter adaptation. As demonstrated in our extended analysis, adjusting the tile size from the default 64 to 32 or 16 enables more fine-grained workload distribution, recovering PE efficiency to 94% even for the most challenging matrix patterns. This parameter tuning can be automated through a lightweight performance model that analyzes matrix sparsity patterns and selects optimal (tile size, PE count) configurations prior to execution.

The TCOO framework supports dynamic selection from multiple pre-characterized parameter sets, allowing the system to automatically match the computational topology to the specific sparsity/size characteristics of each problem instance. This adaptive capability ensures that the low efficiency observed in fixed configurations represents only a temporary state during the parameter exploration phase, not a fundamental limitation of the architecture. Our results confirm that with proper parameter selection, TCOO maintains high resource utilization across diverse matrix structures while preserving the energy efficiency advantages of the quantized computation approach.

Comprehensive scalability analysis across diverse graph structures reveals consistently high PE efficiency

Supplementary Table 9: Module-level resource breakdown for the P4 design (LUTs, Registers (FFs), BRAM, DSPs). BRAM count includes both BRAM18 and BRAM36.

| Module     | Total LUTs | Logic LUTs | LUT RAMs | SRLs | FFs   | BRAM | DSPs | % of Total LUTs |
|------------|------------|------------|----------|------|-------|------|------|-----------------|
| 1 PE       | 18664      | 16072      | 2592     | 0    | 14994 | 0    | 0    | 17.4%           |
| 4 PEs      | 74735      | 64367      | 10368    | 0    | 59967 | 0    | 0    | 70.0%           |
| Dispatcher | 334        | 334        | 0        | 0    | 147   | 0    | 0    | 0.3%            |
| GE         | 3815       | 3303       | 0        | 512  | 2709  | 0    | 96   | 3.5%            |
| VectorRAM  | 561        | 265        | 296      | 0    | 4     | 48   | 0    | 0.5%            |

Supplementary Table 10: Power consumption under different Gset instances (sparsity levels).

| Instance | Sparsity | Dynamic Power (W) | Static Power (W) | Total Power (W) |
|----------|----------|-------------------|------------------|-----------------|
| G31      | 0.50%    | 6.632             | 1.237            | 7.869           |
| G34      | 0.10%    | 6.506             | 1.237            | 7.743           |
| G35      | 0.29%    | 6.458             | 1.237            | 7.695           |
| G43      | 1.00%    | 6.558             | 1.237            | 7.795           |

(> 95%) for large-scale problems spanning matrix sizes from 256 to 8192 nodes and sparsity levels from 1% to 40%, as systematically validated in Supplementary Figure 13(b). The architecture demonstrates remarkable robustness: random graphs maintain near-optimal efficiency (99.0-100%) across all configurations, while power-law graphs achieve 91-100% efficiency depending on size and sparsity. For the most challenging skewed-block patterns, strategic tile-size optimization enables dramatic efficiency improvements over larger tile configurations (65% in 64x64, and 81% in 32x32), achieving over 99% efficiency with 16x16 tiles, effectively overcoming the inherent load-balancing challenges of irregular matrices. This parametric adaptability, combined with sustained high performance across the entire design space, confirms the architecture’s practical viability for real-world optimization problems exhibiting diverse topological characteristics and scale variations.

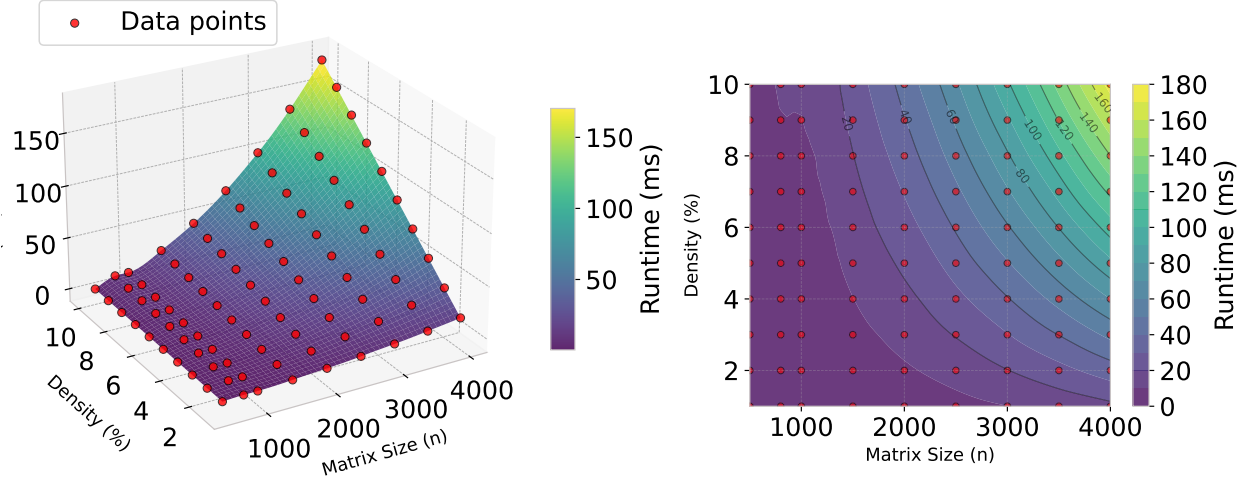
The results collectively demonstrate that our tile-based sparse matrix approach overcomes fundamental limitations of both **MA** and **K2000/K4000**. By achieving superior frequency, efficiency, and flexibility simultaneously, the architecture establishes a new paradigm for FPGA-based Ising processing that could enable previously intractable optimization problems.

**Latency analysis.** For latency, its relationship with matrix sparsity and size has been characterized both empirically and analytically. As supported by our results (e.g., Supplementary Figure 14), the end-to-end latency exhibits a clear and predictable dependence on both the matrix dimension and its sparsity pattern.

Theoretical analysis of our hardware architecture reveals that the dominant component of latency scales with the total number of non-zero elements processed. Since the total number of elements in an  $n \times n$  matrix is  $n^2$ , and the density  $D$  determines the proportion of non-zero elements, the effective workload scales as  $n^2 \cdot D$ . Consequently, the overall latency  $L$  can be modeled as:

$$L \approx kn^2D$$

where  $k$  is a hardware- and implementation-dependent constant encompassing factors such as processing element (PE) pipeline depth, memory access latency, and the fixed per-tile overhead in our TCOO format.



Supplementary Figure 14: End-to-end latency exhibits a joint dependence on matrix size and density. The latency scales quadratically with matrix dimension  $n$  and linearly with density (sparsity  $S$ ), following the approximate relationship  $L \propto n^2 \cdot S$ . The 3D surface (left) and its contour projection (right) visualize this relationship across synthetic matrices.

Our experimental data validate this model. The latency shows an approximately linear dependence on the number of non-zero elements ( $\text{NNZ} = n^2 \cdot S$ ), which is a direct function of both matrix size and sparsity. This quadratic dependence on  $n$  and linear dependence on  $S$  explains the observed trends across different graph instances and sparsity patterns in our evaluation.

**Disentangling TCOO Speedup.** To comprehensively evaluate the individual contributions of speed up lead by TCOO’s key innovations, we conducted a systematic ablation study that disentangles the effects of the tiled format, load balancing, and hardware scheduling. Which isolates the performance impact of each architectural component on representative graph instances:

**Config A (Baseline):** TCOO with software-based hazard resolution and naive load balancing.

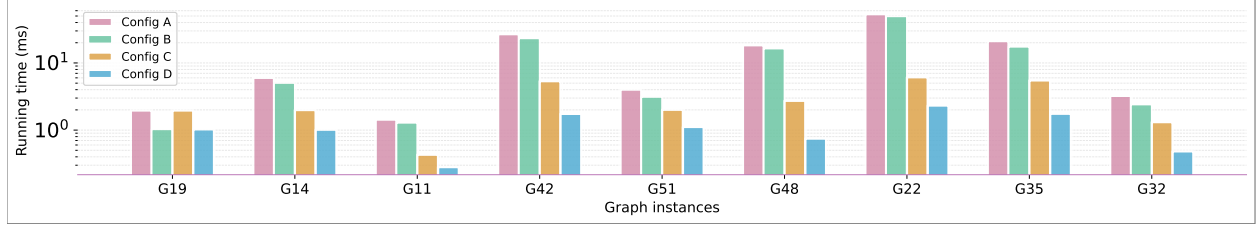
**Config B:** TCOO with software-based hazard resolution and bin-packing load balancing.

**Config C:** TCOO with hardware hazard handling enabled and naive load balancing.

**Config D:** TCOO with bin-packing load balancing and hardware hazard handling enabled.

The results, summarized in Supplementary Figure 15, reveal that each component contributes significantly to the overall performance. Config C achieved an average **2.5x** speedup over the baseline Config A, while Config D delivered a remarkable **65.7%** average reduction in execution time. The bin-packing algorithm alone provided **1.2x** acceleration by effectively balancing computational workload across PEs, while hardware hazard resolution accelerates process through eliminating graph pre-processing time by **59%**. Most notably, the full TCOO configuration demonstrated synergistic effects, where the combined optimizations yielded performance gains exceeding the sum of individual improvements, validating our integrated architectural approach.

Beyond quantization improvements, TCOO introduces fundamental architectural innovations through its explicitly decoupled memory hierarchy. This design strategically separates matrix storage (off-chip DDR) from vector data (on-chip BRAM), providing a first-principle solution to the memory-wall problem. Empirical validation through ablation studies reveals that this architectural foundation enables synergistic optimizations.



Supplementary Figure 15: Contribution of TCOO Components and hardware acceleration; Config A, baseline that uses the software-based rescheduling strategy and naive load balancing, without further optimization; Config B adds bin-packing load balancing; Config C uses hardware hazard handling instead of software and remains naive load balancing, highlighting the benefit of hardware-based conflict resolution; Config D further incorporates bin-packing load balancing, representing the complete TCOO architecture. The figure illustrates the execution time across multiple benchmark graphs.

These coordinated enhancements demonstrate that TCOO represents a distinct architectural approach that fundamentally rethinks sparse matrix computation rather than incrementally adapting existing formats.

## Supplementary References

- [1] Tatsumura, K., Dixon, A. R. & Goto, H. Fpga-based simulated bifurcation machine. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 59–66 (IEEE, 2019).
- [2] Goto, H., Tatsumura, K. & Dixon, A. R. Combinatorial optimization by simulating adiabatic bifurcations in nonlinear hamiltonian systems. *Science advances* **5**, eaav2372 (2019).
- [3] Goto, H. *et al.* High-performance combinatorial optimization based on classical mechanics. *Science Advances* **7**, eabe7953 (2021).
- [4] Paolini, G. V. & Brozolo, G. R. D. Data structures to vectorize cg algorithms for general sparsity patterns. *BIT Numerical Mathematics* **29**, 703–718 (1989).
- [5] Peters, A. Sparse matrix vector multiplication techniques on the ibm 3090 vf. *Parallel Computing* **17**, 1409–1424 (1991).
- [6] Saad, Y. Sparskit: a basic tool kit for sparse matrix computations, version 2. <http://www.cs.umn.edu/saad/software/SPARSKIT/sparskit.html> (1994).
- [7] Gao, J., Liu, B., Ji, W. & Huang, H. A systematic literature survey of sparse matrix-vector multiplication. *arXiv preprint arXiv:2404.06047* (2024).
- [8] Dave, S. *et al.* Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE* **109**, 1706–1752 (2021).
- [9] Yan, S., Li, C., Zhang, Y. & Zhou, H. yaspmv: Yet another spmv framework on gpus. *Acm Sigplan Notices* **49**, 107–118 (2014).
- [10] Tang, W. T., Tan, W. J., Goh, R. S. M., Turner, S. J. & Wong, W.-F. A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu. *IEEE Transactions on Parallel and Distributed Systems* **26**, 2373–2385 (2014).
- [11] Bian, H., Huang, J., Dong, R., Liu, L. & Wang, X. Csr2: a new format for simd-accelerated spmv. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 350–359 (IEEE, 2020).
- [12] Li, Y. *et al.* Vbsf: a new storage format for simd sparse matrix-vector multiplication on modern processors. *The Journal of Supercomputing* **76**, 2063–2081 (2020).
- [13] Monakov, A., Lokhmotov, A. & Avetisyan, A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*, 111–125 (Springer, 2010).
- [14] Barbieri, D., Cardellini, V., Fanfarillo, A., Filippone, S. *et al.* Three storage formats for sparse matrices on gpgpus. (2015).
- [15] Nagasaka, Y., Nukada, A. & Matsuoka, S. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on gpu. *Procedia Computer Science* **80**, 131–142 (2016).
- [16] Yang, W., Li, K. & Li, K. A parallel computing method using blocked format with optimal partitioning for spmv on gpu. *Journal of computer and system sciences* **92**, 152–170 (2018).
- [17] Dang, H.-V. & Schmidt, B. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science* **9**, 57–66 (2012).
- [18] D’Azevedo, E. F., Fahey, M. R. & Mills, R. T. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science*, 99–106 (Springer, 2005).

- [19] Yuan, L., Zhang, Y., Sun, X. & Wang, T. Optimizing sparse matrix vector multiplication using diagonal storage matrix format. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, 585–590 (IEEE, 2010).
- [20] Godwin, J., Holewinski, J. & Sadayappan, P. High-performance sparse matrix-vector multiplication on gpus for structured grid computations. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 47–56 (2012).
- [21] Choi, J. W., Singh, A. & Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM sigplan notices* **45**, 115–126 (2010).
- [22] Liu, W. & Vinter, B. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, 339–350 (2015).
- [23] Kourtis, K., Karakasis, V., Goumas, G. & Koziris, N. Csx: an extended compression format for spmv on shared memory systems. *ACM SIGPLAN Notices* **46**, 247–256 (2011).
- [24] Kestur, S., Davis, J. D. & Chung, E. S. Towards a universal fpga matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 9–16 (IEEE, 2012).
- [25] Bell, N. & Garland, M. Efficient sparse matrix-vector multiplication on cuda. Tech. Rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation (2008).
- [26] Maringanti, A., Athavale, V. & Patkar, S. B. Acceleration of conjugate gradient method for circuit simulation using cuda. In *2009 International Conference on High Performance Computing (HiPC)*, 438–444 (IEEE, 2009).
- [27] Tang, W. T. *et al.* Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 136–145 (IEEE, 2015).
- [28] Zou, Y. & Lin, M. Massively simulating adiabatic bifurcations with fpga to solve combinatorial optimization. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 65–75 (2020).
- [29] Qin, E. *et al.* Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 58–70 (IEEE, 2020).
- [30] Liu, B. & Liu, D. Towards high-bandwidth-utilization spmv on fpgas via partial vector duplication. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 33–38 (2023).
- [31] University, S. Gset. <https://web.stanford.edu/yyye/yyye/Gset/>.
- [32] NVIDIA. Nvidia cusparse library. <https://docs.nvidia.com/cuda/archive/12.0.0/index.html> (2023).
- [33] Anzt, H. *et al.* Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software (TOMS)* **48**, 1–33 (2022).
- [34] Anzt, H. *et al.* Optimizing krylov subspace solvers on graphics processing units. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, 941–949 (IEEE, 2014).
- [35] Intel. Intelmkl. <https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html> (2023).
- [36] Si, J. *et al.* Energy-efficient superparamagnetic ising machine and its application to traveling salesman problems. *Nature Communications* **15**, 3457 (2024).

- [37] Magar, K., Bharathan, S. & Banerjee, U. pc-cop: An efficient and configurable 2048-p-bit fully-connected probabilistic computing accelerator for combinatorial optimization. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–7 (IEEE, 2024).
- [38] Yue, W. *et al.* A scalable universal ising machine based on interaction-centric storage and compute-in-memory. *Nature Electronics* **7**, 904–913 (2024).
- [39] Chen, Y. *et al.* Programmable optoelectronic ising machine for optimization of real-world problems. *Light: Science & Applications* **15**, 6 (2026).
- [40] Yin, X. *et al.* Ferroelectric compute-in-memory annealer for combinatorial optimization problems. *Nature Communications* **15**, 2419 (2024).